Technology
Arts Sciences
TH Köln

Technische Hochschule Köln
Fakultät für Informations-, Medien- und Elektrotechnik

M A S T E R A R B E I T

Vergleichende Analyse der Test- und Wartbarkeit in Xtext-Projekten am Beispiel einer TDL-Implementierung

vorgelegt an der TH Köln
im Studiengang Technische Informatik

von
Martin Sebastian Schulze
Matrikelnummer: [redacted]

Referent: Prof. Dr.-Ing. Georg Hartung
Korreferent: Prof. Dr. Hans Wilhelm Nissen

Köln, 15. Oktober 2018

Mit meinem Verhältnis zur Realität ist alles in bester Ordnung. Ich lasse es alle vierzehn Tage vorschriftsmäßig warten.

Zaphod Beeblebrox

Zusammenfassung

Domänenspezifische Sprachen gewinnen seit einigen Jahren zunehmend an Bedeutung. Xtext ist eine sogenannte *Language Workbench*, mit der solche Sprachen schnell entwickelt werden können. Neben der Sprachinfrastruktur wird eine inzwischen weit fortgeschrittene Integration in die IDE Eclipse erzeugt und es können optional ein Plug-in für IntelliJ und ein Webeditor erstellt werden. Der Ansatz ist dabei, dass der oder die Codegeneratoren direkt mit dem Abstract Syntax Tree arbeiten. In dieser Arbeit wird gezeigt, wie ein Domänenmodell in eine Xtext-Sprache integriert werden kann und wie Test- und Wartbarkeit davon profitieren. Besondere Beachtung finden, gegeben durch das Projektumfeld, die Anforderungen durch Funktionale Sicherheit.

Abstract

Domain-specific languages have become increasingly significant in recent years. With the so-called *Language Workbench* Xtext this languages can be developed quickly. In addition to the language infrastructure, a by now very advanced integration into the IDE Eclipse is created and an optional plug-in for IntelliJ and a web editor can be created. The approach is a code generator(s) work directly with the Abstract Syntax Tree. This thesis decribes a way of integrating a Domain Model into a Xtext language and how testability and maintainability benefit from it. Given the project environment, special attention is paid to requirements made by functional safety.

Danksagung

An dieser Stelle möchte ich mich bei Prof. Dr.-Ing. Georg Hartung für die Betreuung dieser Masterarbeit als Referent, einige sehr hilfreiche Hinweise auf Literatur und die langjährige Unterstützung bedanken. Außerdem danke ich Prof. Dr. Hans Wilhelm Nissen dafür, dass er diese Arbeit als Korreferent betreut hat.

Danke auch an Dr. Tobias Krawutschke und Gerhard Babatunde Faluwoye, die mich neu für das Thema Hardware-Tests fasziniert haben, Irmgard Tietz-Lassotta und Beate Jahn für Korrektur und Housing sowie meinen Eltern und Alexandra für Unterstützung, viel Geduld und manchen Tritt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

ATE Automated Test Equipment / Automatic Test Equipment

AST Abstract Syntax Tree

DSL Domain Specific Language

DUT Design Under Test

EBNF Extended Backus-Naur-Form

EMF Eclipse Modeling Framework

EMOF Essential Meta Object Facility

ERD Entity Relationship Diagramm

ETSI European Telecommunications Standards Institute

FQN Fully Qualified Name

IDE Integrated Development Environment

MOF Meta Object Facility

OCL Object Constraint Language

OMG Object Management Group

TDL Test Description Language

TTCN-3 Testing and Test Control Notation

UML Unified Modelling Language

VCD Value Change Dump

VHDL Very High Speed Integrated Circuit Hardware Description Language

1. Einleitung

Domänenspezifische Sprachen haben in den letzten Jahren zunehmend an Bedeutung gewonnen. Spätestens seit der Veröffentlichung von Fowlers Buch „Domain Specific Languages“[1] 2010 ist das Thema unter diesem Namen bekannt, existiert haben solche Sprachen aber schon viel länger. Tatsächlich hat Fowler diesen Begriff schon in einem Bliki-Artikel 2005 genutzt.[2] Beispiele für Domain Specific Languages (DSLs) sind *Regular Expressions*, die in dieser Arbeit verwendete Sprache *graphviz* und sogar als interne DSL die API von Hamcrest, welches zum Testen eingesetzt wird.

Mit zunehmender Popularität wuchs das Bedürfnis einer Integration in die zur Entwicklung verwendeten Integrated Development Environments (IDEs). Neben stabilen und leistungsfähigen Parser-Generatoren wie ANTLR entstanden sogenannte Language Workbenches. Eines davon, Xtext, erzeugt inzwischen eine sehr umfangreiche Integration in Eclipse, deren hervorstechendstes Merkmal ein vollständiger Editor mit Syntax Highlighting, Anzeige von Warnungen und Fehlern an den richtigen Zeilen sowie eine automatische Formatierung des Programmcodes ist.

In mehreren Abschlussarbeiten am Institut für Nachrichtentechnik an der TH Köln wurde Xtext bereits genutzt. Eine der DSLs wurde im Rahmen des Forschungsprojektes „Entwicklung eines Testautomaten zur Untersuchung der Zulassungsfähigkeit von Software-gesteuerten Baugruppen im Vergleich zu Hardware Baugruppen unter dem Aspekt der funktionalen Sicherheit“ entwickelt. Hierbei handelt es sich um eine Implementierung der Test Description Language (TDL), welche vom European Telecommunications Standards Institute (ETSI) als Europäischer Standard normiert wurde.[3,4]

[1] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Signature Series. Boston: Pearson Education, 2010.

[2] Martin Fowler. *FluentInterface*. 20. Dez. 2005. URL: `https://martinfowler.com/bliki/FluentInterface.html`.

[3] ETSI. *ETSI ES 203 119: Methods for Testing and Specification (MTS); The Test Description Language (TDL); Specification of the Abstract Syntax and Associated Semantics*. Version 1.1.1. Apr. 2014.

[4] ETSI. *ETSI ES 203 119-1: Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 1: Abstract Syntax and Associated Semantics*. Version 1.4.1. Mai 2018.

1.1. Motivation

Die Grundlagen dieser Implementierung wurde von Lauber in seiner Masterarbeit gelegt.[5] Später wurde sie von Lüth[6] sowie vom Autor dieser Arbeit im Rahmen des Forschungsprojektes erweitert. Im Fokus stand dabei immer die Erweiterung der Sprache sowie die Anpassung an Projektbedürfnisse. Durch Architekturentscheidungen, die ganz zu Beginn getroffen wurden, wurde frühzeitig der Weg zu einem ausreichend modularen System verbaut und es wurden keine Tests geschrieben.

Durch die enorme Komplexität der Sprache und des Codegenerators ist es selbst nach mehreren Jahren intensiver Auseinandersetzung mit dem System nicht mehr zu übersehen, was eine kleine Änderung der Sprachsyntax an notwendigen Änderungen im Generator nach sich zieht. Es ist also erforderlich, erste Schritte auf dem Weg zu einer besseren Modularität des Gesamtsystems zu gehen um eine bessere Test- und Anpassbarkeit zu erzielen.

1.2. Zielsetzung

Um eine größere Modularität der TDL-Implementierung zu erreichen, soll ein sogenanntes Domänenmodell eingeführt werden. Statt wie bisher im Codegenerator direkt den Abstract Syntax Tree (AST) zu benutzen, soll dieser nun mit dem Domänenmodell arbeiten. Da dieses Vorgehen bei Xtext unüblich scheint, sollen gleichzeitig Muster erarbeitet werden, wie die Umwandlung von AST zum Domänenmodell geschehen kann und wie die einzelnen Schritte, abweichend von der Literatur, nun getestet werden können.

Neben der konkreten Anwendung im Rahmen der TDL soll gezeigt werden, dass das Verfahren die Entwicklung von komplexen Sprachen verbessern kann. Außerdem sollen die Möglichkeiten des Testens anhand der Literatur und aktueller Normen bewertet werden. Zum Schluss wird gezeigt, dass das Verfahren Änderungen an der Sprachsyntax vereinfacht.

1.3. Gliederung der Arbeit

Die Arbeit ist in vier Teile gegliedert. In Kapitel 2 werden die Grundlagen erarbeitet, die zum Verständnis der Problematik und der vorgestellten Lösung nötig sind. Darauf folgt in Kapitel 3 eine umfassende Darstellung von Ausgangszustand, den Änderungen und den Auswirkungen auf die Tests anhand einer sehr einfachen

[5] David Lauber. "Konzept und Entwicklung einer Testbeschreibungssprache und eines Systems zur Generierung von Testbenches und Testvektoren für einen Testautomaten". Masterarbeit. Fachhochschule Köln, 18. Juni 2015.

[6] René Lüth. "Erzeugung von Testfällen mit einem DSL-basierten Generator auf Basis der ETSI-TDL für einen digitalen Testautomaten". Bachelorarbeit. TH Köln, 22. Feb. 2017.

Beispielsprache, die Fowler auch als Beispiel genutzt und deren Syntax in Xtext als Beispielprojekt enthalten ist. Kapitel 4 wendet die gewonnen Erkenntnisse exemplarisch auf die TDL an. In Kapitel 5 wird bewertet, ob die Einführung eines Domänenmodells tatsächlich Test- und Wartbarkeit erhöht.

Um die Übersicht zu behalten, werden in den meisten Listings nur die relevanten Zeilen im Text eingefügt. Auslassungen werden dabei durch drei aufeinanderfolgende Punkte ... dargestellt. Zur besseren Einordnung wird bei Listings, welche Codeänderungen darstellen, jeweils vor und nach der Änderung eine unveränderte Zeile hinzugefügt. Die vollständigen Listings sind, soweit für das Verständnis der Arbeit relevant, in Anhang A abgedruckt. Vor allem in Kapitel 4 macht es Sinn, nur sehr kleine Ausschnitte des Codes zu zeigen.

Der gesamte Quellcode befindet sich auf dem beigelegten Datenträger. Zu beachten ist, dass dies nur eine Momentaufnahme zum Zeitpunkt der Fertigstellung dieser Arbeit ist. Darum macht ein kompletter Ausdruck sicherlich keinen Sinn, da Programmcode seit langem nicht mehr als statisch betrachtet werden kann. Längst unterliegt er einem ständigen Prozess von Wartung (Fehlerreparaturen), Pflege (Refactoring) und Erweiterung.

1.4. Verwendete Software

Für diese Arbeit wurde die verwendete Software zu Beginn des Bearbeitungszeitraumes ausgewählt. Updates der Minor Versions wurden durchgeführt, jedoch keine Updates der Major Versions. Genutzt wurde folgende Software:

- Java: OpenJDK 1.8.0_181
- Eclipse Version: Oxygen.3a Release (4.7.3a)
  Build id: 20180405-1200
- EMF Xcore Version: 1.5.0.v20170613-0242
- Xtext Version: 2.12.0.v20170519-1412
- Xtend SDK Version: 2.12.0.v20170519-1412
- EclEmma 3.1.1.201809121651
- Infinitest 5.2.0
- PlantUML Version 1.2018.11
- graphviz version 2.40.1 (20161225.0304)

Entwickelt wurde auf einem Ubuntu 18.04 64bit Linux.

2. Grundlagen

Im Folgenden sollen die Grundlagen dargestellt werden, die für das Verständnis der Arbeit nötig sind. Dazu erfolgt zuerst eine genauere Einordnung in den Kontext des Forschungsprojektes. Anschließend werden die verwendeten Tools kurz beschrieben und einige strukturelle Probleme angesprochen. Zum Schluss wird die von mir postulierte Änderung in der Architektur dargestellt und in die richtige Meta-Ebene des Themas „Modellierung“ eingeordnet.

2.1. Automated Test Equipment

Die Normen zur Funktionalen Sicherheit[7,8,9] stellen sehr hohe Ansprüche an die Qualität elektronischer Komponenten und deren Dokumentation. Diese können bei komplexen Systemen nur durch automatisiertes Testen von Teil- und Gesamtsystemen erreicht werden.

Anders als viele Softwaresysteme können Hardwaresysteme meist nicht oder nur sehr eingeschränkt über Debugging-Schnittstellen mit einem handelsüblichen PC getestet werden. In der Regel müssen sowohl in der Entwicklung als auch in der Produktion zur Qualitätssicherung teure und komplexe Geräte genutzt werden um Tests durchzuführen. Die Probleme reichen von der Konnektierung des Design Under Test (DUT) über elektrische Anpassungen, die Testausführung bis zur Datenübertragung und Testauswertung. Diese Anlagen werden unter dem Begriff Automated Test Equipment / Automatic Test Equipment (ATE) zusammengefasst.

2.2. Testautomat

Wie schon im Abschnitt 1.1 dargestellt, wurde an der TH Köln im Rahmen eines Forschungsprojektes die TDL implementiert. Mit dem ebenfalls in diesem Projekt entstandenen Testautomaten kann ein Test, welcher in der TDL formuliert wurde, auf echter Hardware ausgeführt werden. Der Testautomat soll das Back-to-Back-Testen von Bahnkomponenten erleichtert werden, die zu ersetzen sind. Ausgehend von einem Satz Testvektoren zeichnet der Testautomat die Reaktionen

[7] IEC. *IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems.* Apr. 2010.

[8] DIN. *DIN EN 50128; VDE 0831-128:2012–03: Bahnanwendungen - Telekommunikationstechnik, Signaltechnik und Datenverarbeitungssysteme - Software für Eisenbahnsteuerungs- und Überwachungssysteme; Deutsche Fassung EN 50128:2011.* März 2012.

[9] ISO. *ISO 26262: Road vehicles – Functional safety.* Nov. 2011.

der Hardware auf. Diese werden dann mit den erwarteten Reaktionen verglichen. Krawutschke et al haben eine gute Übersicht über den Testablauf gegeben.[10]

Einer der zentralen Ansätze im oben genannten Forschungsprojekte war die Einführung der TDL als Quelle für alle Stimuli, erwarteten Reaktionen und der Testbench, die zum Überprüfen der Ergebnisse benötigt wird. Diese ist als DSL mit Xtext implementiert worden. Dadurch wird das fehleranfällige Erstellen von Testvektoren, Ergebnissen, Testbench und Konfiguration vermieden.

2.3. DSLs

Spätestens seit der Veröffentlichung von Martin Fowlers Buch "Domain Specific Languages" gewinnt der Begriff DSL zunehmende Bekanntheit. Abgegrenzt werden muss der Begriff DSL von vollwertigen Programmiersprachen auf der einen und einfachen Konfigurationssprachen auf der anderen Seite.

Die DSL ist deutlich ausdrucksstärker als eine einfache Konfigurationssprache. Gleichzeitig ist der Anwendungsbereich, im Gegensatz zu einer Programmiersprache, stark begrenzt.

Außerdem muss zwischen textuellen und graphischen DSLs unterschieden werden. Textuelle DSLs haben einen oft unterschätzten Vorteil gegenüber graphischen DSLs: sie lassen sich mit den in der Softwareentwicklung bekannten Tools verarbeiten. Dazu gehören einfache Texteditoren, Quellcodeverwaltung, Änderungsverfolgung und Tools zum Code-Review. Graphische DSLs haben allerdings den Vorteil, dass sie für Menschen deutlich besser zu erfassen sind. Nicht ohne Grund wurden in vielen Bereichen graphische Darstellungen oder Sprachen entworfen. Als Beispiele können Zustandsautomaten, Unified Modelling Language (UML), Petrinetze, Entity Relationship Diagramme (ERD) und Wahrheitstabellen genannt werden. Für die TDL wurde eine textuelle Darstellung gewählt, da diese in der Norm vorgegeben ist.

[10] Tobias Krawutschke u. a. “Test automation for reengineered modules using test description language and FPGA”. In: *embedded world Conference 2018 – Proceedings*. embedded world Conference 2018 – Proceedings: WEKA FACHMEDIEN GmbH, 2018.

2.4. Xtext

In den letzten Jahren sind Sprachframeworks immer ausgereifter geworden. Xtext, seit über zehn Jahren in aktiver Entwicklung (der erste Commit auf Github datiert auf Mai 2008[11]), ist neben JetBrains MPS eines der ausgereiftesten. Es basiert auf Eclipse, nutzt sehr intensiv die Möglichkeiten des Eclipse Modeling Framework (EMF) und Antlr als Parser-Generator. Die Sprachdefinition selbst erfolgt in einer Extended Backus-Naur-Form (EBNF), es können direkt Plugins für Eclipse, IntelliJ IDEA und einen Web-Editor erzeugt werden. In dieser Arbeit wird die Version Xtext 2.12.0 verwendet, welche in Eclipse Oxygen.3a enthalten ist.

2.4.1. Domänenmodell

Wie in der Einleitung vorgeschlagen, soll ein Domänenmodell dazu genutzt werden, die Codegeneratoren der TDL von den syntaktischen Eigenheiten der Sprache selbst zu entkoppeln. Dieses Modell ist mehr als nur ein Datenmodell. Evans definiert das Domänenmodell als „A system of abstractions that describes selected aspects of a domain and can be used to solve problems related to that domain.“[12] Gleichzeitig verbindet er es direkt mit der sogenannten *ubiquitous language*, die er definiert als „A language structured around the domain model and used by all team members within a bounded context to connect all the activities of the team with the software.“[13]

Es geht also nicht nur darum, Daten für den Transport zu modellieren sondern ein sprachliches Abbild der Problemdomäne zu entwickeln und im Modell festzuhalten. Dadurch wird das Nutzen des Modells, hier primär in den Codegeneratoren, vereinfacht. Die *ubiquitous language*, übersetzbar als Allgegenwärtige Sprache, dient gleichzeitig dazu, Missverständnissen im Entwicklungs-Team vorzubeugen. Die Namen der Objekte, ihrer Inhalte und Beziehungen untereinander etablieren als Domänenmodell also auch die sprachliche Grundlage in der Kommunikation sowohl im Team als auch mit Domänenexperten.

[11] Sven Efftinge. *initial commit*. 9. Mai 2008. URL: `https://github.com/eclipse/xtext/commit/e6d6a78063ae6ea5d80b25ace7b4416b0e169654` (besucht am 10. 10. 2018).

[12] Eric Evans. *Domain-Driven Design Reference. Definitions and Pattern Summaries*. März 2015. URL: `https://domainlanguage.com/wp-content/uploads/2016/05/DDD_Reference_2015-03.pdf`, S. vi.

[13] Evans, *Domain-Driven Design Reference*, S. vi.

2.4.2. Ecore und Xcore

Wird Eclipse zur Entwicklung genutzt und ein Datenmodell soll erstellt werden, bietet es sich an, die Tools des EMF[14] zu nutzen, zumal Xtext darauf schon aufbaut. Die Vorgehensweise ist, dass das Modell als Instanz eines Meta-Modells erstellt wird. Daraus werden dann Java-Klassen, Interfaces, eine Klasse mit Fabrik-Methoden und einige Infrastruktur-Klassen generiert. Der Vorteil ist eine deutlich übersichtlichere und kompakte Darstellung als reiner Java-Code und ein reduzierter Wartungsaufwand.

Verwendet man Ecore als Meta-Modell, so bieten sich mehrere Editoren zur Modellierung an, darunter ein sehr einfacher, listenorientierter und ein grafischer Editor (EcoreTools). Die Alternative, welche in dieser Arbeit verwendet wurde, heißt Xcore und nutzt eine textuelle Darstellung. Diese lässt sich gut versionieren und bietet eine sehr kompakte Syntax zur schnellen Entwicklung des Datenmodells.

2.4.3. Projektstruktur und Build

Die momentan von Xtext genutzte Projektstruktur weist einige Besonderheiten auf. Berücksichtigt wird hier nur die Struktur, welche mit dem Projekt-Wizard erstellt wurde, das heißt ein Eclipse-Projekt mit Maven-Build. Wenn kein Eclipse-Plugin erzeugt werden soll, lässt sich Xtext auch mit der bei Maven üblichen Projektstruktur nutzen. Die erste Besonderheit ist, dass das Gesamtprojekt inzwischen in neun einzelne Eclipse-Projekte aufgeteilt wurde. Das Haupt-Projekt bekommt den im Projekt-Wizard angegebenen „Projektnamen“, die anderen Projekte bekommen ein zusätzliches Postfix.

Projektname	Bedeutung
de.th_koeln.nt.tdl.parent	Parentprojekt
de.th_koeln.nt.tdl	Hauptrojekt
de.th_koeln.nt.tdl.tests	Tests für das Hauptprojekt
de.th_koeln.nt.tdl.ui	UI-Komponenten
de.th_koeln.nt.tdl.ui.tests	Tests für UI-Komponenten
de.th_koeln.nt.tdl.ide	Eclipse-Editor
de.th_koeln.nt.tdl.target	Spezifikation der Zielplattform
de.th_koeln.nt.tdl.feature	Eclipse-Feature, welches als Plugin installiert werden kann
de.th_koeln.nt.tdl.repository	Update-Site

Tabelle 1: Xtext-Projektstruktur am Beispiel des TDL-Projekts

Tabelle 1 zeigt am Beispiel der TDL, welche Projekte angelegt werden. Im unteren Abschnitt der Tabelle sind die Projekte aufgeführt, die eclipse-spezifische

[14] Eclipse Foundation. *Eclipse Modeling Framework (EMF)*. 2018. URL: `https://www.eclipse.org/modeling/emf/` (besucht am 10. 10. 2018).

Aufgaben übernehmen. Hier macht die gewählte Einteilung Sinn, da die einzelnen Projekte sehr unterschiedliche Aufgaben erfüllen.

Die Projekte im oberen Abschnitt der Tabelle könnten für einen Maven-Build besser aufgeteilt werden. Die Tests sollen laut Martin Spiller[15] im zugehörigen Projekt abgelegt werden. Maven sieht hier eine passende Ordnerstruktur vor. Die bei Xtext gezeigte Abweichung widerspricht stark dem Prinzip „Convention over Configuration“, auf welches die Maven-Referenz an prominenter Stelle hinweist.[16] Die Trennung der UI-Komponenten vom Sprachen-Kern macht allerdings Sinn, da diese für einen Standalone-Compiler nicht benötigt werden.

Die Ordnerstruktur in den einzelnen Eclipse-Projekten weicht wiederum deutlich von der Struktur ab, die von Maven favorisiert wird. So liegen die Quelldateien im Ordner `/src` anstatt `/src/main/java`. Außerdem enthalten die Ordner `/src-gen` und `/xtend-gen` generierte Quellcode-Dateien, die im Build zusätzlich berücktsichtigt werden müssen.

Eine letzte Abweichung von der Konvention ist, dass die Testausführung nicht in der Build-Phase `test` sondern in der Build-Phase `integration-test` stattfindet. Dies muss beim Aufruf von Maven und bei der Konfiguration eines Build-Servers berücksichtigt werden.

Diese Abweichungen von der Maven-Konvention führen in Summe zu einem ziemlich unübersichtlichen Build. Die fehlenden Standard-Quellverzeichnisse führen zu Warnungen von Maven mit dem Inhalt

```
1 [WARNING] Directory /home/martin/svn/tdl/trunk/de.th\_koeln
  ↪ .nt.tdl.parent/de.th\_koeln.nt.tdl/src/test/java is empty. Can't
  ↪ process
```

die allerdings nicht bedeuten, dass ein Fehler vorliegt. Eine Anpassung an die Konvention wäre hier angebracht.

[15] Martin Spiller. *Maven 3: Konfigurationsmanagement mit Java*. 1. Aufl. Verfasserangabe: Martin Spiller ; Quelldatenbank: FHBK-x ; Format:marcform: print ; Umfang: 356 S. : Ill. ; 978-3-8266-9118-8 Pb. : EUR 29.95 (DE); 3-8266-9118-0 Pb. : ca. EUR 29.95 (DE). Heidelberg u.a.: Mitp, 2011. URL: `http://digitale-objekte.hbz-nrw.de/storage/2011/09/24/file%5C_12/4297430.pdf`, S. 63

[16] o.V. *Maven: The Complete Reference*. o.J. URL: `http://books.sonatype.com/mvnref-book/pdf/mvnref-pdf.pdf` (besucht am 02. 05. 2018), S. 2

2.5. Template Engines, Xtend

Zu Beginn dieser Arbeit war geplant, eine Template-Engine mit externen Templates zu nutzen. Eine für Java-Projekte interessante Wahl wäre Apache Velocity gewesen,[17] es hätte aber auch andere Alternativen gegeben.[18] Aufgrund der Unsicherheit, ob eine dieser Template-Engines mit Ecore unproblematisch zusammenarbeitet, da Xtend in Xtext massiv Verwendung findet und wegen der sehr übersichtlichen Möglichkeiten der Funktionalen Programmierung auf Listen wurde entschieden, doch Xtend zu benutzen. Für eine Beschreibung der Sprache sei an dieser Stelle auf die offizielle Dokumentation verwiesen.[19]

2.6. Übersicht der Architektur

Die bisherige Architektur der TDL-Implementierung entspricht dem Xtext-Standard, wie Abbildung 1 verdeutlicht. Das Problem an dieser Architektur ist, dass die Komplexität so nicht mehr zu handhaben ist.

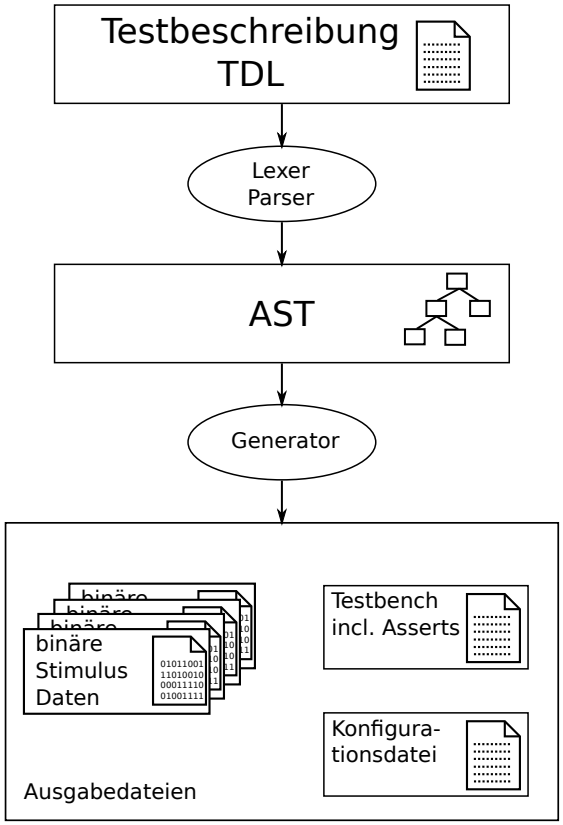


Abbildung 1: Architektur ohne Domänenmodell

[17] The Apache Software Foundation. *What is Velocity?* 2016. URL: https://velocity.apache.org/ (besucht am 10. 10. 2018).

[18] Miro Wegner. *A Review of Java Template Engines*. 2. Dez. 2016. URL: https://dzone.com/articles/template-engines-at-one-spring-boot-and-engines-se (besucht am 10. 10. 2018).

[19] o.V. *Xtend Documentation*. o.J. URL: https://www.eclipse.org/xtend/documentation/101_gettingstarted.html.

Obwohl die Sprachdefinition (als EBNF) nur 82 Regeln enthält, sind diese auf knapp 800 Zeilen verteilt. Einige dieser Regeln werden nicht genutzt, es sind offensichtliche Fehler vorhanden und es lässt sich nur schwer eine Übersicht gewinnen. Daraus wird das Ecore-Model für den AST erzeugt, welches aus 81 Interfaces und den zugehörigen Klassen besteht. Hier wäre eine Modularisierung dringend angeraten.

Ebenso ist der Generator fast nicht zu überblicken. Abgesehen von zwei kleinen Generatoren, welche später entstanden sind und ausschließlich Konfigurationsdateien erzeugen, besteht dieser aus einer einzigen, 1216 Zeilen langen Xtend-Datei, welche vier Dateien erzeugt:

1. Die Stimuli für die Hardware als Value Change Dump (VCD)-Datei.
2. Die erwartete Antwort der Hardware auch als VCD-Datei.
3. Einen VCD-Player als Very High Speed Integrated Circuit Hardware Description Language (VHDL)-Datei, welcher die Signalverläufe aus den VCD-Dateien als VHDL-Signale mit richtigem Timing zur Verfügung stellt.
4. Eine VHDL-Testbench, welche mit dem VCD-Player dazu in der Lage ist, die VCD-Dateien zu überprüfen

Das Problem ist, dass nur die zweite Datei in einer eigenen Methode generiert wird (eine eigene Klasse mit mehreren internen Methoden wäre noch besser), die anderen drei jedoch durch denselben Code. Wichtig ist auch, dass die Dateien exakt in der richtigen Reihenfolge erzeugt werden, da dieser Code abhängig davon, welche Datei gerade erzeugt wird, einen internen Zustand ändert. Dieser besteht hauptsächlich aus Indizes über bestimmten gefilterten Listen sowie einigen Variablen und Flags.

Der Programmcode selber besteht aus mehreren Methoden (hier hat der Versuch stattgefunden, eine gewisse Modularität aufzubauen), die sich gegenseitig rekursiv abhängig vom internen Zustand aufrufen, der sich wiederum ständig ändert. Das macht es fast unmöglich, wirklich zu verstehen, welcher Code welche Ausgabe erzeugt und unter welchen Bedingungen er das tut. Die ständigen Iterationen über kompliziert gefilterte Listen des AST tun ihr übriges.

Es macht an dieser Stelle keinen Sinn, Teile des Codes einzufügen, da dieser aufgrund der Zeilenlänge komplett unlesbar umgebrochen würde. Die längste Zeile ist tatsächlich 523 Zeichen lang. Einige Screenshots in den Abbildungen 2 und 3 sollen jedoch genügen, ein Gefühl dafür vermitteln, wie enorm das Problem ist.

Abbildung 2: TDL-Generator-Code

Abbildung 3: TDL-Generator-Code zu breit für den Bildschirm

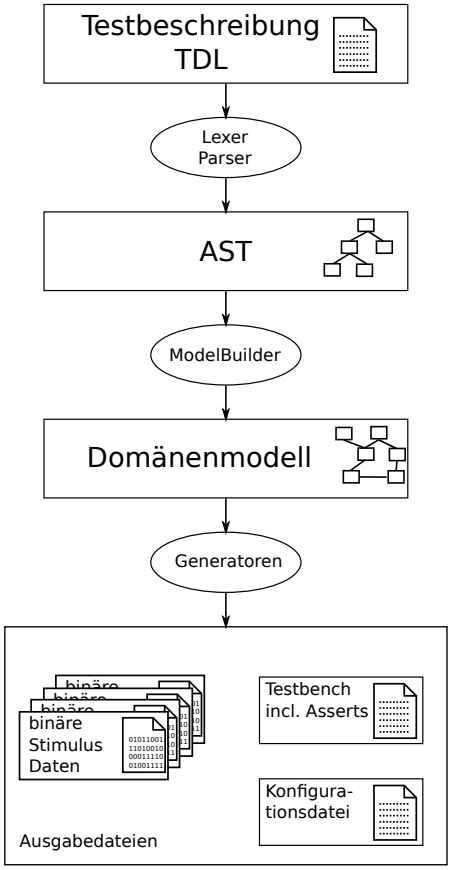


Abbildung 4: Architektur mit Domänenmodell

Es ist wahrscheinlich unnötig zu erwähnen, dass in der bisherigen Codebasis keinerlei Tests abgesehen von generierten Tests für den AST vorliegen. Diese können mit einem aktuellen Setup nicht kompiliert werden, da sie auf JUnit 3 basieren. Außerdem ist nichts implementiert abgesehen vom Setup und Teardown der Testfixture. Es sind also keine Tests im eigentlichen Sinne.

Ein letztes Problem, welches sich durch die Komplexität des Generators ergeben hat, ist die wirklich miserable Performance des Editors beim Bearbeiten des Generators. Selbst auf einem High-End-PC legt dieser beim Editieren „Denkpausen“ von etwa zwei Sekunden ein, die auf weniger leistungsfähigen Maschinen bis zu zehn Sekunden lang werden. In dieser Zeit reagiert der Eclipse-Editor gar nicht. Das führt in der Praxis dazu, dass es einfacher ist, Änderungen in einem externen Editor vorzubereiten, diese einzufügen und nur einmal auf das Ende der Pause zu warten.

Ziel ist es also, durch eine Änderung der Architektur gemäß Abbildung 4 in einem ersten Schritt die Generatoren vom AST und damit von der Sprachdefinition zu trennen. Diese Trennung geschieht durch das Domänenmodell, welches wenn möglich direkt modular sein sollte. Eine Trennung ist hier um jeden sogenannten *Bounded Context* herum möglich. Diesen definiert Evans als „A description of a boundary (typically a subsystem, or the work of a particular team) within which

a particular model is defined and applicable.“[20] Eine sinnvolle Grenze dafür liegt zwischen der Testkonfiguration und den eigentlichen Instanzen konkreter Tests.

Später kann dann die Sprachdefinition und damit das Modell des AST modularisiert werden. Das wird für eine bessere Übersicht sorgen. Außerdem können die Fehler und ungenutzten Regeln dort repariert werden, da durch eine kleine syntaktische Änderung keine umfangreiche Änderung des Generators nötig wird.

Exkurs: Modelle und Metamodelle

In dieser Arbeit wird der Begriff Domänenmodell konsequent falsch genutzt, da dies in der Literatur auch so geschieht und anders auch wenig Sinn macht. Konsequent müsste man vom Domänen-Meta-Modell sprechen, dessen Instanzen dann Domänenmodelle sind. Abbildung 5 verdeutlicht diesen Sachverhalt und ordnet gleichzeitig alle Modelle in eine Hierarchie ein. An dessen Spitze steht das Essential Meta Object Facility (EMOF)-Metamodell. Als Teil der Meta Object Facility (MOF) wurde diese von der Object Management Group (OMG) eingeführt und normiert.[21] Das MOF-Metamodell ist gleichzeitig ein MOF-Modell und beschreibt sich auf diese Weise selber. So wird einer unendlichen Hierarchie der Meta-Modellierung vorgebeugt.

[20] Evans, *Domain-Driven Design Reference*, S. vi.

[21] Object Management Group. *OMG Meta Object Facility (MOF) Core Specification*. Nov. 2016. URL: `https://www.omg.org/spec/MOF` (besucht am 11. 10. 2018).

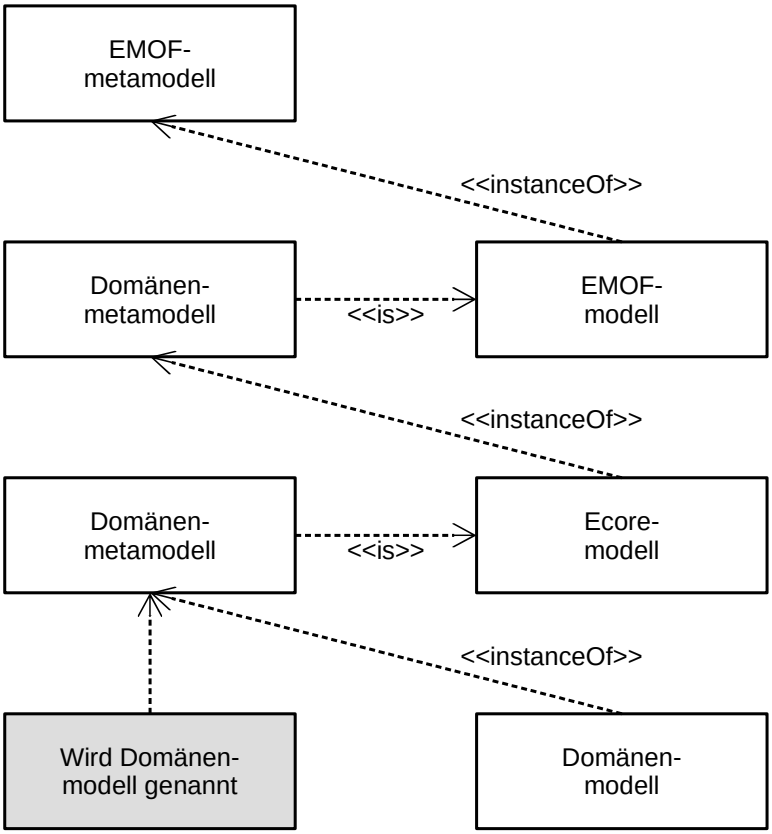


Abbildung 5: Meta-Ebenen der Modelle

3. DSL für Endliche Automaten

In seinem Buch „Domain Specific Languages“[22] stellt Martin Fowler eine DSL zur Beschreibung von Endlichen Automaten vor. Diese Sprache nutzt Fowler an vielen Stellen in seinem Buch, um einzelne Aspekte in der Entwicklung einer DSL beispielhaft darzustellen. Sie ist dabei so einfach, dass sie sich auch ideal zur Veranschaulichung der Problematik eignet, die hier gezeigt werden soll.

3.1. Einführung

Die Entwickler von Xtext haben Fowlers Sprache zur Definition Endlicher Automaten bereits als eins der Beispielprojekte in Eclipse bereitgestellt. Das Projekt enthält neben der EBNF wie jedes neue Xtext-Projekt Platzhalter für alle Sprach- und Editor-Funktionen sowie einen Generator, der Java-Code erzeugt. Leider ist der Beispielgenerator recht komplex und eignet sich daher nicht gut dazu, die in Abschnitt 1.1 dargestellte Problematik zu demonstrieren.

In Abschnitt 3.2 wird ein deutlich einfacherer Codegenerator vorgestellt, der aus einem Endlichen Automaten eine Graphviz-Datei (genauer eine DOT-Datei) erzeugt. Aus dieser kann mit Graphviz eine ansprechende Darstellung des Automaten generiert werden. Zuerst wird hier der schlecht testbare Ansatz verfolgt, um das Problem konkret zu zeigen. Der Fokus liegt auf der Entwicklung des Generators, das heißt auf Änderungen, die sich typischerweise im Entwicklungsprozess ergeben.

Darauf aufbauend wird im Folgenden der Unterschied zu einem entkoppelten Domänenmodell dargestellt und es werden die nötigen Entwurfsmuster aus der Literatur herangezogen. Zum Schluss des Kapitels werden die Auswirkungen auf die Struktur der Tests und die Testabdeckung untersucht.

Listing 1 zeigt den einfachen Endlichen Automaten, der im Folgenden als Beispiel genutzt wird um die Entwicklung des Generators zu demonstrieren.

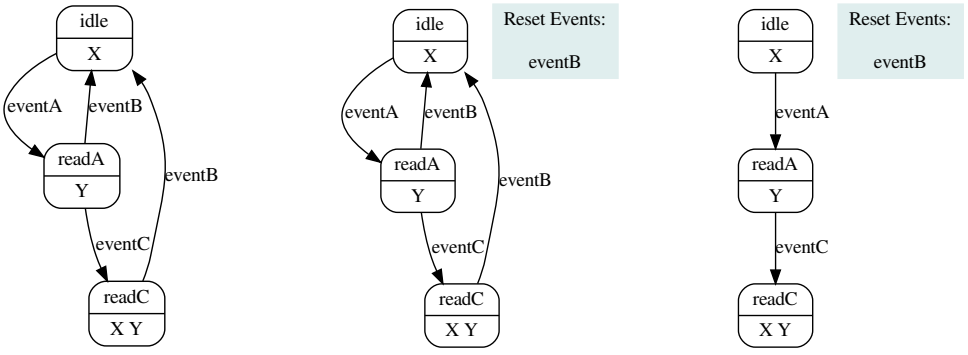
3.2. Codegenerator ohne Domänenmodell

Der einfache Code-Generator wurde, wie im Xtext-Universum üblich, in Xtend geschrieben. Listing 2 zeigt den Generator in einer ersten Version. In Abbildung 6 wird die Entwicklung der Darstellung vom ersten Entwurf bis zum korrekten Graphen deutlich.

[22] Fowler, *Domain-Specific Languages*, S. 3 ff.

```
1   events
2   eventA a
3   eventB b
4   eventC c
5   end
6
7   resetEvents
8   eventB
9   end
10
11  commands
12  OutputX X
13  OutputY Y
14  end
15
16  state idle
17  actions {OutputX }
18  eventA => readA
19  end
20
21  state readA
22  actions {OutputY }
23  eventB => idle
24  eventC => readC
25  end
26
27  state readC
28  actions {OutputX OutputY}
29  eventB => idle
30  end
```

Listing 1: State machine example



Von links nach rechts: Graph ohne Reset-Events, Graph mit Angabe der Reset-Events, Graph um Reset-Events bereinigt

Abbildung 6: Generierter Graph

```
1  package de.th_koeln.nt.fowler.generator
2
3  import org.eclipse.emf.ecore.resource.Resource
4  import org.eclipse.xtext.generator.AbstractGenerator
5  import org.eclipse.xtext.generator.IFileSystemAccess2
6  import org.eclipse.xtext.generator.IGeneratorContext
7  import de.th_koeln.nt.fowler.fowler.State
8
9  class FowlerGenerator extends AbstractGenerator {
10
11    override void doGenerate(Resource resource, IFileSystemAccess2 fsa,
       ↪  IGeneratorContext context) {
12       fsa.generateFile("fsm.dot", graphviz(resource))
13    }
14
15    def String graphviz(Resource resource) {
16       '''
17       digraph finite_state_machine {
18          size="8,5"
19          node [shape = record, style = rounded];
20          «FOR state :
    ↪  resource.allContents.toIterable.filter(typeof(State))»
21          «state.name»[label="{«state.name»|«FOR action :
    ↪  state.actions»«action.code» «ENDFOR»}"];
22          «ENDFOR»
23
24          «FOR state :
    ↪  resource.allContents.toIterable.filter(typeof(State))»
25          «FOR transition : state.transitions»
26          «state.name» -> «transition.state.name» [ label =
    ↪  "«transition.event.name»" ];
27          «ENDFOR»
28          «ENDFOR»
29
30       }
31       '''
32    }
33 }
```

Listing 2: Graphviz-Generator

Im Vergleich mit der Darstellung, die Fowler zeigt,[23] fällt am ersten Graphen in Abbildung 6 auf, dass die Angabe der Reset-Events fehlt. Diese führen automatisch in den Zustand *idle* zurück. In diesem Beispiel ist das Event *eventB* das einzige Reset-Event.

[23] Fowler, *Domain-Specific Languages*, S. 5

```
...
7  import de.th_koeln.nt.fowler.fowler.State
8  import de.th_koeln.nt.fowler.fowler.Statemachine
9
10 class FowlerGenerator extends AbstractGenerator {
...
32             «ENDFOR»
33
34         node [shape = box, style = filled, color=azure2]
35         resetEvents[label="Reset Events:\n«FOR resetEvent :
       ↪ resource.allContents.toIterable
36             .filter(typeof(Statemachine))
37             .last.resetEvents»\n«resetEvent.name»«ENDFOR»"]
...
```

Listing 3: Graphviz-Generator, erste Erweiterung

```
...
28             «FOR state :
       ↪ resource.allContents.toIterable.filter(typeof(State))»
29             «FOR transition : state.transitions.filter[
30                 !
                   ↪ resource.allContents.toIterable.filter(typeof(Statemachine))
31                 .last.resetEvents.contains(it.event)
32             ]»
33             «state.name» -> «transition.state.name» [ label =
       ↪ "«transition.event.name»" ];
...
```

Listing 4: Graphviz-Generator, zweite Erweiterung

Listing 3 zeigt die Erweiterungen, welche zur Anzeige der Reset-Events führen (Abbildung 6 Mitte), Listing 4 die Änderung, welche zur korrekten Darstellung führt (Abbildung 6 rechts). Hier werden die Events, welche zum Zustand *Idle* zurück führen, im Graphen nicht mehr explizit dargestellt.

Auffällig ist, dass die Ausdrücke in den Schleifenköpfen der FOR-Schleifen immer länger, komplexer und weniger verständlich werden. In Verbindung mit den nötigen Anforderungen an die Formatierung des generierten Textes entstehen entweder überlange Zeilen wie Zeile 35 in Listing 3 oder Konstrukte wie in den Zeilen 29 bis 32 in Listing 4, welche die hierarchische Struktur der Schleifen fast unlesbar machen.

An diesem einfachen Beispiel wird erkennbar, was in größeren Sprachen immer deutlicher zutage tritt: Die Struktur der Sprache entspricht selten der Struktur der Problemdomäne. Wird dann, wie hier, direkt auf dem AST generiert, muss an vielen Stellen das Modell gefiltert, neu sortiert und anderweitig bearbeitet werden.

Geschieht das wie hier direkt im Template des Codegenerators, wird dieses sehr unübersichtlich.

Im Beispiel wird in einem Abschnitt mit der Überschrift *resetEvents* eine Liste von Events angegeben, die im Graphen nicht explizit dargestellt werden sollen. Solche Ausnahmen von der Regel führen zu Filter-Anweisungen. In anderen Fällen müssen mehrere Listen miteinander verbunden und neu sortiert werden. Oft ist im generierten Text eine gleichbleibende Reihenfolge derselben Elemente an unterschiedlichen Stellen gewünscht. Alle diese Besonderheiten der Domäne müssen konsistent über das gesamte Template beachtet werden.

Parr unterscheidet[24] drei Pattern für die sogenannte „translation“, also das Übersetzen von einer Sprache in eine andere. Sowohl der „Syntax-Directed Translator“ als auch der „Rule-Based Translator“ generieren ihre Ausgabe direkt aus der Eingabesprache und sind im Xtext-Umfeld nicht geeignet. Der „Model-Driven Translator“, beschrieben von Terence Parr ab Seite 280, ist also das Pattern, welches hier eingesetzt wird. Auch Martin Fowler[25] nutzt in der Regel ein Domänenmodell.

Zur Verwirrung sowohl in der Bachelorarbeit des Autors[26] als auch in der Masterarbeit von David Lauber[27] mag die nicht ganz klare Nomenklatur in der Xtext-Dokumentation[28] beigetragen haben. Zum Zeitpunkt des Abrufs wurde dort im Tutorial vom *domain model* gesprochen, obwohl eigentlich der AST gemeint ist. Tatsächlich verschwimmen hier die Grenzen zwischen AST und Domänenmodell etwas, da der von Xtext generierte AST sehr viele zusätzliche Objektreferenzen enthält. In der ordentlichen Dokumentation wird inzwischen aber korrekt der AST als das von Xtext generierte Datenmodell genannt.

3.3. Domänenmodell

Es ist also für eine Sprache, deren Struktur nicht klar mit der Problemdomäne übereinstimmt, nötig, ein Domänenmodell aufzustellen. Fowler spricht gar davon, dass er sich nur sehr wenige Anwendungen einer DSL ohne Domänenmodell vorstellen kann.[29] Seiner Meinung nach ermöglicht dies erst eine Trennung von

[24] Terence Parr. *Language implementation patterns : create your own domain-specific and general programming languages*. Raleigh, N.C: Pragmatic Bookshelf, 2010, S. 276.
[25] Fowler, *Domain-Specific Languages*.
[26] Martin Schulze. “Defining a Domain Specific Language for the Configuration of SoPCs Based on the OpenRISC Platform”. Bachelorarbeit. Fachhochschule Köln, Juli 2013.
[27] Lauber, “Konzept und Entwicklung einer Testbeschreibungssprache und eines Systems zur Generierung von Testbenches und Testvektoren für einen Testautomaten”.
[28] o.V. *Xtext Dokumentation*. o.J. URL: `https://www.eclipse.org/Xtext/documentation/index.html` (besucht am 03. 08. 2018).
[29] Fowler, *Domain-Specific Languages*, S. 162.

Parser und Semantik.[30] Das deckt sich mit den Erfahrungen, die mit der TDL gemacht wurden.

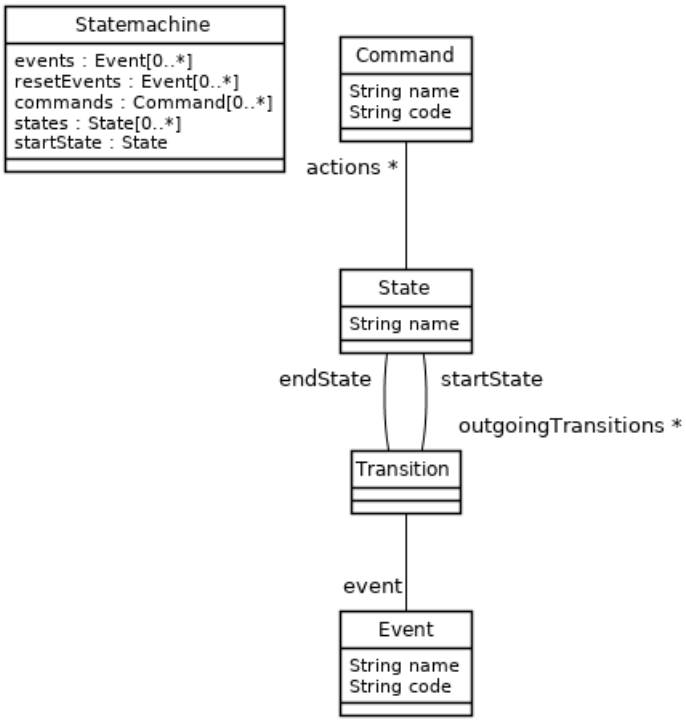


Abbildung 7: Domänenmodell der DSL

Abbildung 7 zeigt eine Möglichkeit der Modellierung. Es gibt sicherlich noch andere sinnvolle Ansätze, die hier jedoch nicht diskutiert werden sollen.

Xtext benutzt für den AST ein Ecore-Modell. Es bietet sich also an, für das Domänenmodell dasselbe Framework zu benutzen. Im Umfeld von Ecore sind diverse Tools zum Erstellen und Visualisieren von Datenmodellen entstanden. Eines davon ist Xcore, eine DSL zum Beschreiben von Datenmodellen.

Listing 5 zeigt die Umsetzung des Domänenmodells in Xcore. Zusätzlich zu dem eigentlichen Domänenmodell werden Indizes angelegt, sogenannte Symboltabellen. Diese ermöglichen das Iterieren über alle Instanzen einer Klasse.

3.4. Tree-Walker und andere Pattern

In der Literatur werden einige Designpattern zum Instanziieren des Domänenmodells dargestellt. Während Fowler in einem kurzen Abschnitt den „tree-walk“ beispielhaft in Kombination mit dem ANTLR-Framework darstellt,[31] ihn aber nur in der einleitenden Grafik des Kapitels so benennt,[32] widmet Parr dem Infor-

[30] Fowler, *Domain-Specific Languages*, S. 162.
[31] Fowler, *Domain-Specific Languages*, S. 288 ff.
[32] Fowler, *Domain-Specific Languages*, S. 281.

```
1  package de.th_koeln.nt.fowler.model
2
3  class Statemachine {
4     contains Event[] events
5     contains Event[] resetEvents
6     contains Command[] commands
7     contains State[] states
8     refers State startState
9  }
10
11 class Event {
12    String name
13    String code
14 }
15
16 class Command {
17    String name
18    String code
19 }
20
21 class State {
22    String name
23    refers Transition[] outgoingTransitions opposite startState
24    refers Command[] actions
25 }
26
27 class Transition {
28    refers Event event
29    refers State startState opposite outgoingTransitions
30    refers State endState
31 }
```

Listing 5: Xcore-Modell der Fowler-DSL

mationsgewinn aus dem AST ein ganzes Kapitel.[33] Parr beschreibt vier Muster steigender Komplexität, die dazu geeignet sind, einen AST in ein Domänenmodell zu überführen (oder direkt Code zu generieren, was hier, wie bereits dargestellt, aber nicht sinnvoll ist).

Der *Embedded Heterogeneous Tree Walker* kann im Xtext-Umfeld nicht genutzt werden. Dazu müssten die Klassendefinitionen des Domänenmodells erweitert werden. Das ist zwar mit den „Extension Methods“ in Xtend möglich, wäre dann allerdings äquivalent zum zweiten Muster.

Tree Grammar und *Tree Pattern Matcher* sind zwar mächtige Pattern, die auch von ANTLR, welches von Xtext benutzt wird, unterstütz werden. Xtext abstrahiert

[33] Terence Parr. *The Definitive ANTLR Reference*. English. Raleigh, NC: Pragmatic Bookshelf, 2007, S. 99 ff.

aber so stark über dem ANTLR-Framework, dass eine Modifikation hier einen zu tiefen Eingriff in den generierten Code bedeuten würde. An diesem Punkt wäre eine Erweiterung seitens der Xtext-Entwickler wünschenswert. Vor allem der Tree Pattern Matcher würde Strukturerkennung im AST deutlich vereinfachen.

Der *External Tree Visitor* ist also momentan das einzig sinnvolle Muster im Zusammenspiel mit Xtext. Die im Codegenerator von Xtext aufgerufene Methode `doGenerate(Resource resource, ...) { ... }` bekommt als ersten Parameter den AST in einem Container vom Typ *Resource*. Dieser bietet mit der Methode *getContent* den Zugriff auf das Wurzelelement des AST.

Dieses ist, obwohl in der Regel nur ein einzelnes Element, vom Typ *EList<EObject>*. Diese Klassen sind Teil des ECore-Models, in dem der AST definiert ist. Über die Methode *eContents* des in der Liste enthaltenen EObject gelangt man an die Kind-Elemente. Eine Typ-Bestimmung für den nötigen Typecast in einem *Tree Visitor* ist mit den ECore-Mitteln einfach.

Trotzdem wird dieser Ansatz hier nicht weiter verfolgt. Bevor auf eine mit Xtend sehr elegante Lösung für kleine Domänenmodelle eingegangen wird, sollen noch zwei alternative Ansätze dargestellt werden.

3.5. Xtext model mapping

Xtext erlaubt, neben der automatischen Generierung des ECore-Modells für den AST, die Nutzung eines von vorneherein selbst erstellen ECore- (oder Xcore-) Modells. Die einzelnen Sprachelemente werden dann in der EBNF-Definition der Sprachsyntax direkt auf dieses Datenmodell „gemapt“. Solange die grundsätzliche Struktur der Sprache mit derjenigen des Datenmodells übereinstimmt, ist dieser Weg durchaus praktikabel.

Im Falle der Statemachine-Beschreibung, die hier beispielhaft genutzt wird, wäre das sicherlich das Mittel der Wahl. Die Syntax der Test Description Language (TDL) unterscheidet sich allerdings strukturell deutlich von dem Domänenmodell. Vor diesem Hintergrund wird auch für die deutlich einfachere Sprache eine andere Lösung bevorzugt.

3.6. Model-to-Model Transformation

Eine weitere Möglichkeit, aus dem AST das Domänenmodell zu instanziieren, bietet das Eclipse-Projekt „Model-to-Model Transformation“, welches selbst ein Unterprojekt des EMF ist.[34]

[34] o.V. *Webseite: Model-to-Model Transformation in Eclipse*. o.J. URL: `https://projects.eclipse.org/projects/modeling.mmt` (besucht am 20. 09. 2018).

3.7. Domänenmodell erzeugen mit Xtend

Xtend bietet mit *Extension Methods*, der Vorwegnahme der Java Stream API und *Lambda Expressions* leistungsfähige Mittel, um das Domänenmodell aufzubauen.

3.7.1. Modularisierung

An dieser Stelle ist es wichtig, sich Gedanken über die Modularität zu machen. Die Versuchung ist gerade ganz zu Beginn der Entwicklung einer neuen DSL groß, den gesamten Code, der zum Generieren aller Ergebnisse nötig ist, in der von Xtext generierten Generator-Klasse zu schreiben. Das führt jedoch schnell zu einer sehr langen Datei mit mehreren Tausend Zeilen stark gekoppeltem Code. Dazu kommt, dass der Teil des Programmcodes, welcher der Datenverarbeitung dient, reiner Xtend-Code ist, während der Teil, welcher die Artefakte generiert, größtenteils aus Template-Ausdrücken besteht. Sollen Dateien in unterschiedlichen Sprachen generiert werden, wird es dann völlig unübersichtlich.

Mindestens genauso schlecht wie die für menschliche Leser kaum noch zu überblickende Komplexität ist aber die Testbarkeit. So müsste ein Test in einer solchen Architektur ein vollständiges und gültiges Dokument der DSL eingeben und die daraus generierten Artefakte auf Korrektheit überprüfen. Das bedeutet einerseits hauptsächlich String-Vergleiche, die weder in Java noch in Xtend wirklich angenehm zu programmieren sind. Außerdem zeigt ein fehlschlagender Test nicht, wo der Fehler aufgetreten ist.

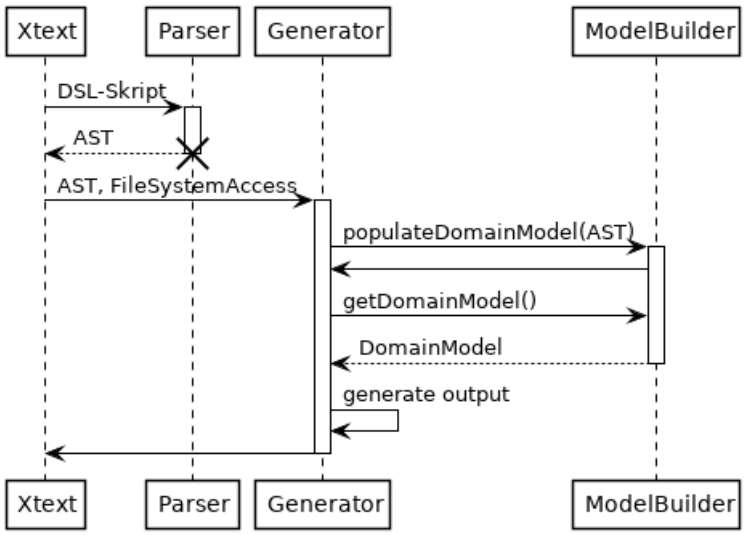


Abbildung 8: Sequenzdiagramm des Ablaufs mit Domänenmodell

Es ist also essentiell, den Aufbau des Domänenmodells vom Rest des Generators zu trennen. Xtext bietet die Möglichkeit, den Schritt von DSL-Text zum AST einzeln zu testen. Es ist also ein Modul zu erstellen, welches aus dem AST das

Domänenmodell erzeugt und gut einzeln testbar ist. Abbildung 8 veranschaulicht den Ablauf eines Kompiliervorgangs mit Domänenmodell. Der letzte Aufruf des Generators auf sich selbst wäre bei komplexeren Sprachen und vor allem, wenn mehrere Artefakte generiert werden sollen, entsprechend auf externe Generator-Klassen aufzuteilen.

3.7.2. Implementierung

Der *External Tree Visitor* wird also als Xtend-Klasse `ModelBuilder` implementiert. Während die ModelFactory im Konstruktor erzeugt wird, geschieht das Erzeugen des eigentlichen Domänenmodells in der Methode `populateDomainModel(Resource ast)`, die in Listing 6 dargestellt wird. Das bietet die Möglichkeit, für mehrere Generatoren jeweils eigene Instanzen des Domänenmodells zu erzeugen.

```
14  class ModelBuilder {
...
29      def populateDomainModel(Resource ast) {
30          this.statemachine = modelFactory.createStatemachine
31
32          ast.resetEvents().forEach[preprocessResetEvents(it)]
33          ast.filterFor(Event).forEach[addEvent(it as Event)]
34          ast.filterFor(Command).forEach[addCommand(it as Command)]
35          ast.filterFor(State).forEach[addState(it as State)]
36          ast.filterFor(State).forEach[wireTransitions(it as State)]
37          setStartState(ast.filterFor(State).head as State)
38      }
```

Listing 6: ModelBuilder.populateDomainModel

Zuerst wird eine Instanz des Wurzel-Elements vom Domänenmodell erzeugt. Danach folgen einige Durchläufe durch den AST, in denen jeweils Daten extrahiert und im Domänenmodell eingefügt werden. Dieses Muster wird auch von Fowler beschrieben,[35] kann hier aber deutlich vereinfacht genutzt werden. Die beiden Methoden `resetEvents()` sowie `filterFor()` sind, wie in Listing 7 gezeigt als *Extension Methods* genutzt.

[35] Fowler, *Domain-Specific Languages*, S. 290 ff.

```
14  class ModelBuilder {
...
95      def filterFor(Resource ast, Class<?> instanceClass) {
96          ast.allContents.filter[it.eClass.instanceClass.equals(instanceClass)]
97      }
98
99      def EList<Event> resetEvents(Resource ast) {
100         ast.allContents.toIterable
101             .filter(typeof(de.th_koeln.nt.fowler.fowler.Statemachine))
102             .get(0).resetEvents
103     }
```

Listing 7: ModelBuilder.filterFor und ModelBuilder.resetEvents

Beide Methoden nutzen eine Besonderheit der Klasse `Resource`, die zwar mit der Methode `getContents` Zugriff auf das Wurzelelement des AST bietet, jedoch über `getAllContents` direkt einen `TreeIterator<EObject>` liefert. Dieser abstrahiert vom eigentlichen *Tree Walk*. Mit dem funktionalen Interface der *IteratorExtensions* von Xtend kann dann sehr komfortabel gefiltert werden.

```
14  class ModelBuilder {
...
61      def addCommand(Command command) {
62          val de.th_koeln.nt.fowler.model.Command commandToAdd =
            ↪  modelFactory.createCommand
63          commandToAdd.code = command.code
64          commandToAdd.name = command.name
65          commands.put(commandToAdd.name, commandToAdd)
66          statemachine.commands.add(commandToAdd)
67      }
```

Listing 8: ModelBuilder.addCommand

Als Beispiel einer der einfachen Methoden, an welche die gefilterten Objekte des AST weitergereicht werden, um das Domänenmodell aufzubauen, kann die Methode `addCommand(Command command)`, in Listing 8 dargestellt, dienen. Die Klasse `Command` des Parameters ist aus dem Package des AST importiert, unter anderem um den Aufruf der Methode `filterFor` anschaulich zu machen. Aus diesem Grund kann die lokale Variable `commandToAdd` vom Typ `Command` aus dem Package des Domänenmodells nur über den Fully Qualified Name (FQN) erstellt werden. Darauf müssen die Daten aus dem `Command` AST in das neue Objekt übernommen werden. Zum Schluss wird das neue `Command` einer Symboltabelle und dem Domänenmodell hinzugefügt.

Selbstverständlich können diese Methoden deutlich komplexer werden. Die Methode `addState(State state)` in Listing 9 beispielsweise überträgt auch die `Actions`, welche in einem State ausgeführt werden sollen, in das Domänenmodell. Dazu muss die Symboltabelle der `commands` durchsucht werden (Zeilen 74 f.) nach den `actions`, welche im AST im State als `actions` vorhanden sind (Zeilen 73 und 77). Das funktionale Interface der *IteratorExtensions* und *Lambda Expressions* macht das Filtern und Transformieren von Daten dabei deutlich übersichtlicher.

```
14  class ModelBuilder {
...
69      def addState(State state) {
70          val de.th_koeln.nt.fowler.model.State stateToAdd =
            ↪  modelFactory.createState
71          stateToAdd.name = state.name
72          stateToAdd.actions.addAll(
73              state.actions.map [ original |
74                  commands.filter [ key, value |
75                      key == original.name
76                  ].values
77              ].flatten
78          )
79          states.put(stateToAdd.name, stateToAdd)
80          statemachine.states.add(stateToAdd)
81      }
```

Listing 9: ModelBuilder.addState

Hier werden gleichzeitig die Stärken als auch eine der Schwächen von Xtend bei der Erzeugung des Domänenmodells deutlich. Durch die Abstraktion vom *Tree Walk*, das mächtige Funktionale Interface (welches seit Java 8 mit etwas anderer Syntax Teil der Stream API ist) der *IteratorExtensions* und *Extension Methods* ist Xtend sehr ausdrucksstark. Die ModelBuilder-Klasse wird aber bei größeren Datenmodellen unübersichtlicher und macht eine weitere Modularisierung oder Abstraktion nötig.

3.8. Code-Generator mit Domänenmodell

Das Domänenmodell ist im Generator einfach erstellt und genutzt. Listing 10 zeigt das deutlich. Zuerst wird ein `ModelBuilder` erzeugt. Damit wird aus dem AST das Domänenmodell erstellt. Anschließend können die Zielartefakte erzeugt werden.

Listing 11 zeigt anschaulich, dass auf einem Domänenmodell mit hoher Kohäsion und sinnvoll gewählten Instanz-Listen die Iterationen im Generator ausdrucksstark sind und ohne oder mit sehr viel weniger Listenoperationen auskommen. Ein

solches Datenmodell ist auch dazu geeignet, in einer **Template Engine** wie velocity benutzt zu werden.

```
14  class FowlerGenerator extends AbstractGenerator {
...
16      override void doGenerate(Resource resource, IFileSystemAccess2 fsa,
        ↪  IGeneratorContext context) {
17          val ModelBuilder modelBuilder = new ModelBuilder
18          modelBuilder.populateDomainModel(resource)
19          fsa.generateFile("fsm.dot",
            ↪  graphviz(modelBuilder.getDomainModel()))
20      }
```

Listing 10: Graphviz-Generator, Instanziierung des ModelBuilders

```
14  class FowlerGenerator extends AbstractGenerator {
...
22      def String graphviz(Statemachine statemachine) {
23
24          '''
25          digraph finite_state_machine {
26              size="8,5"
27
28
29
30              node [shape = record, style = rounded];
31              «FOR state : statemachine.states»
32              «state.name»[label="{«state.name»|«FOR action :
    ↪  state.actions»«action.code» «ENDFOR»}"];
33              «ENDFOR»
34
35              «FOR state : statemachine.states»
36              «FOR transition : state.outgoingTransitions»
37              «state.name» -> «transition.endState.name» [ label =
    ↪  "«transition.event.name»" ];
38              «ENDFOR»
39              «ENDFOR»
40
41          node [shape = box, style = filled, color=azure2]
42          resetEvents[label="Reset Events:\n«FOR resetEvent :
    ↪  statemachine.resetEvents»\n«resetEvent.name»«ENDFOR»"]
43
44          }
45          '''
46      }
```

Listing 11: Graphviz-Generator, zweite Erweiterung

3.9. Testen der DSL

Das Testen einer DSL ist eine sehr umfangreiche Aufgabe. Neben den Tests des Compilers auf Korrektheit muss, wenn mit einem Sprachframework wie Xtext gearbeitet wird, auch die Framework-Integration getestet werden. Bei Xtext handelt es sich um Tests des *Formatters*, der Formatierungshilfen zur Verfügung stellt, des *Conntent Assists*, des Editors, der Integration in die IDE - in diesem Falle Eclipse - sowie zusätzlicher Funktionen wie dem Outline-Fenster und dem Wizard zum Erstellen neuer Projekte. Bettini bietet dazu eine gute Übersicht,[36] geht bei den einzelnen Punkten aber leider nicht sehr ins Detail.

Obwohl die vorgenannten Tests wichtig sind, liegt der Fokus dieser Arbeit auf den Änderungen, die durch das Domänenmodell entstehen. Auch das Testen des Parsers und des Generators erklärt Bettini. Dazu zeigt er auch in einem Vortrag, dass es möglich ist, 100% Testabdeckung zu erreichen.[37] Dazu jedoch mehr im Abschnitt 3.10 Testabdeckung.

3.9.1. Testvorbereitung

Tests in Java werden traditionell mit JUnit durchgeführt. Um effizient testen zu können, sollte man einige Klassen so gestalten, dass sie durch das von Xtext verwendete Dependency Injection Framework *guice* zur Verfügung gestellt werden können. Dazu wird die vom Xtext-Framework generierte Klasse `FowlerRuntimeModule` erweitert wie in Listing 12 dargestellt.

```
13 class FowlerRuntimeModule extends AbstractFowlerRuntimeModule {
14
15         def Class<? extends ModelBuilder> bindModelBuilder() {
16             ModelBuilderImpl
17          }
18         def Class<? extends FowlerFactory> bindFowlerFactory() {
19             FowlerFactoryImpl
20          }
21         def Class<? extends ModelFactory> bindModelFactory() {
22             ModelFactoryImpl
23          }
24 }
```

Listing 12: FowlerRuntimeModule.xtend

Zum Ausführen von Tests ist JUnit gut geeignet, es hat sich jedoch gezeigt, dass es wenig ausdrucksstark ist, Tests mit den traditionellen Methoden `Assert.assertTrue()`

[36] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. 2nd Revised edition. Birmingham: Packt Publishing, 31. Aug. 2016.

[37] Lorenzo Bettini. *Testing Xtext Languages*. ab 27;58. 9. Nov. 2015. URL: `https://youtu.be/fTkwFJy3EeM?t=1678` (besucht am 07. 10. 2018).

und `Assert.assertEquals()` zu schreiben. Aus diesem Grund sind die beiden Frameworks *AssertJ* und *Hamcrest* entstanden. Beide bieten eine Methode `assertThat()`, welche in der Regel statisch importiert wird.

AssertJ bietet ein sogenanntes *Fluent Interface*, das heißt, dass mehrere Behauptungen über einem Objekt mit dem Punkt-Operator aneinandergereiht werden. Eine gute Einführung in AssertJ hat Vogel veröffentlicht.[38]

Aufgrund der in einem vorherigen Projekt erworbenen Erfahrung des Autors wurde in dieser Arbeit Hamcrest genutzt. Nachdem erste Tests nicht wie erwartet funktioniert haben, ist ein Problem aufgefallen, welches durch die Benutzung von ECore-Modellen entsteht. Das ECore-Framework stellt eine umfangreiche Infrastruktur bereit, welche die Arbeit mit Datenmodellen massiv erleichtert. Dafür muss man mit einigen Besonderheiten rechnen. Hier handelt es sich um die Implementierung von `equals(Object)` und `hashCode()`. Die folgende Aussage aus der JavaDoc der Klasse EObject erklärt das Problem: „The framework also assumes that implementations will not specialize #equals(Object) (nor #hashCode()) so that "==" can be always used for equality testing; EcoreUtil.equals should be used for doing structural equality testing.“[39] Kreft und Langer erklären in einer Artikelserie, dass das bei Wertklassen ungünstig ist.[40,41,42] Dementsprechend funktionieren die Hamcrest-Matcher nicht, die auf `equals(Object)` basieren.

[38] Lars Vogel. *Testing with AssertJ assertions - Tutorial*. 29. Aug. 2016. URL: `http://www.vogella.com/tutorials/AssertJ/article.html` (besucht am 07. 10. 2018).

[39] IBM Corporation u. a. *Interface EObject*. 2006. URL: `http://download.eclipse.org/modeling/emf/emf/javadoc/2.4.2/org/eclipse/emf/ecore/EObject.html` (besucht am 07. 10. 2018).

[40] Angelika Langer Klaus Kreft. *Objektvergleich Wie, wann und warum implementiert man die equals()-Methode? Teil 1: Die Prinzipien der Implementierung von equals()*. Jan. 2002. URL: `http://www.angelikalanger.com/Articles/EffectiveJava/01.Equals-Part1/01.Equals1.html` (besucht am 07. 10. 2018).

[41] Angelika Langer Klaus Kreft. *Objektvergleich Wie, wann und warum implementiert man die equals()-Methode? Teil 2: Der Vergleichbarkeitstest*. März 2002. URL: `http://www.angelikalanger.com/Articles/EffectiveJava/02.Equals-Part2/02.Equals2.html` (besucht am 07. 10. 2018).

[42] Angelika Langer Klaus Kreft. *Hash-Code-Berechnung Wie, wann und warum implementiert man die hashCode()-Methode? Teil 1: Die Prinzipien der Implementierung von equals()*. Jan. 2002. URL: `http://www.angelikalanger.com/Articles/EffectiveJava/03.HashCode/03.HashCode.html` (besucht am 07. 10. 2018).

```
1  package de.th_koeln.nt.fowler.tests;
...
9  public class EqualsEObject extends TypeSafeMatcher<EObject> {
10
11    private EObject actual;
12
13    public EqualsEObject(EObject actual) {
14       this.actual = actual;
15    }
16
17    @Override
18    public void describeTo(Description description) {
19       description.appendText(actual.toString());
20    }
21
22    @Override
23    protected boolean matchesSafely(EObject item) {
24       if (EcoreUtil.equals(item, actual)) {
25          return true;
26       } else {
27          return false;
28       }
29    }
30
31    @Factory
32    public static EqualsEObject equalsEObject(EObject actual) {
33       return new EqualsEObject(actual);
34    }
35 }
```

Listing 13: Hamcrest-Matcher EqualsEObject

Um mit Hamcrest trotzdem vernünftig Tests auf dem AST und dem Domänenmodell auszuführen wurden zwei eigene Matcher geschrieben. Der erste Matcher in Listing 13 wird dazu benutzt, um zwei Objekte von Unterklassen der Klasse `EObject` auf Gleichheit zu testen. Listing 14 zeigt den zweiten Matcher, der testet, ob ein bestimmtes Objekt einer Unterklasse von `EObject` in einem Iterator vom Typ `Iterator<? extends EObject>` vorhanden ist. Diese Vorgehensweise erlaubt es, beliebige Modellteile auf das Vorhandensein eines solchen Objektes zu prüfen, da jedes `EObject` und auch die `Resource`, welche den gesamten AST enthält, ein Objekt vom Typ `TreeIterator` erzeugen kann, welches den kompletten Teilbaum des jeweiligen Modells enthält.

```
1  package de.th_koeln.nt.fowler.tests;
...
```

```
15 public class ContainsEObject extends BaseMatcher<Iterator<? extends
   ↪  EObject>> {
16
17    private EObject actual;
18
19    public ContainsEObject(EObject actual) {
20       this.actual = actual;
21    }
22
23    @Override
24    public void describeTo(Description description) {
25       description.appendText(actual.toString());
26    }
27
28    @Override
29    public boolean matches(Object object) {
30       if (object instanceof Iterator<?>) {
31          try {
32             @SuppressWarnings("unchecked")
33             Iterator<EObject> list = (Iterator<EObject>) object;
34             Stream<EObject> objectStream = StreamSupport
35                   .stream(Spliterators.spliteratorUnknownSize(list,
                      ↪  Spliterator.NONNULL), false);
36             if (objectStream.anyMatch(item -> EcoreUtil.equals(item,
                   ↪  actual))) {
37                return true;
38             } else {
39                return false;
40             }
41          } catch (ClassCastException e) {
42             return false;
43          }
44
45       }
46
47       return false;
48    }
49
50    @Factory
51    public static ContainsEObject containsEObject(EObject actual) {
52       return new ContainsEObject(actual);
53    }
54
55 }
```

Listing 14: Hamcrest-Matcher ContainsEObject

Mithilfe dieser beiden Matcher lassen sich in Xtend auf einem Ecore-Datenmodell sehr aussagekräftige Tests formulieren. In den folgenden Abschnitten werden die erstellten Tests ausschnittsweise erklärt. Hier liegt der Fokus auf den genutzten

Konzepten und nicht auf einer Vollständigen Darstellung. Vollständige Listings sind wieder im Anhang zu finden.

3.9.2. Testen des Parsers

Der Test des Parsers wurde in zwei unabhängige Xtend-Dateien aufgeteilt. Die eine enthält einige sehr einfache Tests des Parsers, die andere einen fast vollständigen Test eines Automaten, der alle Sprachfunktionen nutzt.

```
22 @RunWith(typeof(XtextRunner))
23 @InjectWith(typeof(FowlerInjectorProvider))
24 class FowlerSimpleParsingTest {
25    @Inject extension ParseHelper<Statemachine>
26    @Inject extension ValidationTestHelper
27
28    @Inject FowlerFactory astFactory
...
90 }
```

Listing 15: FowlerSimpleParsingTest.xtend

Die Klassendefinition des einfachen Tests wird in Listing 15 gezeigt. Zeile 22 zeigt, dass der Test von einem XtextRunnter ausgeführt werden muss. Das gilt für alle Xtext-Tests. Der annotierte `FowlerInjectorProvider` muss in Tests, die das Benutzerinterface testen, durch einen `FowlerUIInjectorProvider` ersetzt werden. In Zeile 25 wird ein `ParseHelper<Statemachine>` als *extension* injiziert, der die Anweisung `parse(CharSequence)` zur Verfügung stellt. Ebenso wird der `ValidationTestHelper` injiziert, welcher die Anweisung `assertNoErrors(EObject)` enthält. Die injizierte `FowlerFactory` wird zum Erzeugen von AST-Elementen genutzt.

Listing 16 zeigt einen der Tests. Obwohl dieser Test sehr einfach gehalten ist, bedarf die genutzte Xtend-Syntax doch einer Erklärung.

```
24  class FowlerSimpleParsingTest {
...
53      @Test
54      def void eventsAreCreated() {
55          '''
56              events
57                  reset R0
58              end
59          '''.parse => [
60              Assert.assertNotNull("Das Result ist null.", it)
61              assertNoErrors
62              assertThat("Event is not created", it.eAllContents,
                ↪  containsEObject(astFactory.createEvent => [code="R0"
                ↪  name="reset"]))
63          ]
64      }
```

Listing 16: FowlerSimpleParsingTest.xtend

Die in den Zeilen 55 bis 59 zwischen den dreifachen Anführungszeichen eingeschlossenen Zeichen sind eine sogenannte Template expression, werden hier aber nur als mehrzeiliges String-Literal genutzt, um das Newline-Zeichen nicht ausschreiben zu müssen. Die Methode `parse` in Zeile 59 hat die Signatur `Statemachine ParseHelper.parse(CharSequence text)`. Dadurch, dass sie als *extension* eingebunden wird, kann man sie ohne den Klassennamen als Statische Methode aufrufen.

Das Anhängen mit dem Punkt-Operator nutzt die Besonderheit von Xtend aus, dass man auf jedem Objekt eine beliebige Methode durch Anhängen aufrufen kann, wen der erste Parameter vom Typ dieses Objekts ist. Auf dem String-Literal wird ein Typecast nach `CharSequence` ausgeführt und das Ergebnis als Parameter an die Methode `parse` übergeben.

Der folgende Pfeiloperator darf nicht mit dem Java-Operator für Lambda-Ausdrücke verwechselt werden. Hier erzeugt der Operator einen sogenannten Lokalen Kontext, welcher den Rückgabewert der Methode `parse` als Variable `it` enthält. Dieser ist nicht, wie im Generator, vom Typ `Resource` sondern der AST vom Typ `Statemachine`, es fehlt also die äußere Hülle. Soll diese auch getestet werden, enthält der `ParseHelper` auch dafür passende Methoden.

Der Aufruf von `assertNoErrors` erfolgt wieder ohne Parameter. Hier wird implizit die Variable `it` übergeben. Dieser Aufruf dient mit der vorherigen Zeile als Schutz vor einem ungültigen AST in der nächsten Zeile.

Dort wird im Aufruf von `assertThat` der Hamcrest-Matcher `containsEObject` genutzt, um zu überprüfen, dass im AST das Event auch angelegt wurde. Das Objekt, mit dem verglichen werden soll, wird mit der `astFactory` produziert und

in einem Lokalen Kontext mit Werten gefüllt. Hier werden auf der Variablen `it` implizit die Methoden `setCode` und `setName` aufgerufen.

Gerade beim Erstellen von komplexen Objekten mit vielen Werten ist ein solcher Aufruf deutlich lesbarer als ein Konstruktor-Aufruf, der hier im Übrigen gar nicht zur Verfügung steht. Auf diese Weise lassen sich auch komplexere Tests auf dem AST formulieren, die lesbar und aussagekräftig bleiben. Im Anhang findet sich dazu das Listing der Testklasse `FowlerComplexParsingTest`.

3.9.3. Testen des ModelBuilder

Wird beim Testen des Parsers Text als Eingabe genutzt, so ist die Eingabe des `ModelBuilder` ein Abstract Syntax Tree. Wird die Sprache komplexer, so wird der Manuelle Aufbau eines AST komplex oder sogar unmöglich, in jedem Fall jedoch sehr unleserlich.

Obwohl es verlockend ist, den AST einfach vom Parser aus einem Stück Code der DSL erzeugen zu lassen, sollte es dringend vermieden werden. Zum einen würde der `ModelBuilder` nicht isoliert getestet, zum anderen erzeugt das manuelle Aufbauen eines AST ein gutes Verständnis der Struktur. So können Probleme in der Sprachsyntax erkannt werden, die durch ein Review der Syntax in EBNF wahrscheinlich nicht auffallen würden.

Wünschenswert ist also ein ModelBuilder, der mit einem unvollständig gefüllten AST zurecht kommt. Eine Überprüfung daraufhin, ob zum Beispiel Pflicht-Elemente vorhanden sind, sollte man daher dem Parser oder Validator überlassen.

Wie in Listing 17 gezeigt beschränkt sich das Testen dann auf das Erzeugen von einzelnen AST-Teilen, das Erstellen des Domänenmodells (im Beispiel Zeile 53) und das anschließende Überprüfen der Elemente im Domänenmodell.

```
28  class FowlerModelBuilderTest {
...
41      @Test
42      def void eventsArePopulatedIntoModel() {
43          val Event e1 = astFactory.createEvent => [name = "Event1"; code
            ↪  = "E1"]
44          val Event e2 = astFactory.createEvent => [name = "Event2"; code
            ↪  = "E2"]
45
46          val Statemachine sm = astFactory.createStatemachine
47          ast.contents.add(sm)
48
49          sm.events.add(e1)
50          sm.events.add(e2)
51          sm.resetEvents.add(e2)
52
```

```
53          modelBuilder.populateDomainModel(ast)
54
55          assertThat("Event is built into the Domainmodel",
56             modelBuilder.domainModel.eAllContents,
57             containsEObject(modelFactory.createEvent => [name = "Event1";
               ↪  code = "E1"]))
58          assertThat("Event is built into the Domainmodel",
59             modelBuilder.domainModel.eAllContents,
60             containsEObject(modelFactory.createEvent => [name = "Event2";
               ↪  code = "E2"]))
61      }
...
```

Listing 17: FowlerModelBuilderTest.xtend

3.9.4. Testen des Generators und Integrationstests

Der Generator wird wieder nach demselben Schema getestet. Zuerst wird das Domänenmodell erstellt, dann wird dieses dem Generator übergeben und das Ergebnis, ein String, wird entsprechend auf Richtigkeit überprüft.

Da hier letztlich nichts Neues passiert, bleibt der Hinweis darauf, dass der Generator per Dependency Injection zur Verfügung gestellt wird. Werden in einem Projekt mehrere Generatoren benötigt, so müssen diese direkt benannt werden oder, sofern sie jeweils ein Interface implementieren, über die Klasse `FowlerRuntimeModule` bekannt gemacht werden.

In jedem Fall sollten auch Integrationstests durchgeführt werden. Nur so besteht die Sicherheit, dass die einzelnen Schritte auch nacheinander korrekt ausgeführt werden. Das Vorgehen entspricht hier dem von Bettini[43] im Abschnitt „Testing code generation“.

3.9.5. Testen des StandaloneSetup

Abschließend folgt ein Test des StandaloneSetup. Dieses wird benötigt, wenn für die Sprache ein Compiler erstellt werden soll, welcher unabhängig von der Eclipse-IDE ausgeführt wird.

Zum Laden einer DSL-Datei benötigt man ein `XtextResourceSet`. Im normalen Eclipse-Ablauf würde man sich dieses vom Dependency Injection Framework zur Verfügung stellen lassen. Da Dependency Injection im Standalone-Betrieb nicht genutzt oder erst aufwändig konfiguriert werden müsste, erstellt man zuerst einen Injektor (Listing 18, Zeile 27).

[43] Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend*, S. 136.

Damit wird dann das `XtextResourceSet` erstellt, welches zum Laden der `XtextResource` genutzt wird (Zeile 29). In dem Moment, wo diese geladen wird, wird das DSL-Skript geparst und in der `Resource` der AST abgelegt.

Für diesen Test reicht es, sich von der `XtextResource` einen `ResourceValidator` zu holen, diesen auszuführen und zu überprüfen, dass die zurückgegebene Liste der Probleme leer ist. Dann hat das Laden der Resource funktioniert.

```
1  package de.th_koeln.nt.fowler.tests.standalone;
...
23 public class FowlerStandaloneTest {
24
25    @Test
26    public void standaloneTest() {
27       Injector injector = FowlerStandaloneSetup.doSetup();
28       XtextResourceSet rs =
         ↪  injector.getInstance(XtextResourceSet.class);
29       Resource resource =
         ↪  rs.getResource(URI.createFileURI("./standalone_test.std"),
         ↪  true);
30       IResourceValidator validator = ((XtextResource)resource)
31             .getResourceServiceProvider().getResourceValidator();
32       List<Issue> issues = validator.validate(
33             resource, CheckMode.ALL, CancelIndicator.NullImpl);
34       assertThat("There are no issues", issues, empty());
35    }
36 }
```

Listing 18: FowlerStandaloneTest.java

Dies entspricht dem Vorgehen von Efftinge[44] zum Kompilieren von Xtext-Sprachen ohne Eclipse. Zarnekow weist darauf hin,[45] dass vor der Validierung noch `EcoreUtil.resolveAll(resource)` aufzurufen ist, um eventuelle Linker-Fehler aufzudecken. Für diesen Test ist dieser Aufruf aber irrelevant, da der einzige wirklich relevante Fehler eine leere Resource wäre.

3.10. Testabdeckung

Spätestens wenn Software für Anwendungen entwickelt wird, bei denen es um Funktionalen Sicherheit geht, werden Qualitätskriterien wichtig. Eines dieser Kriterien ist eine möglichst vollständige Testabdeckung. Wie schon erwähnt behauptet

[44] Sven Efftinge. *How And Why Use Xtext Without The IDE*. 16. März 2016. URL: `https://typefox.io/how-and-why-use-xtext-without-the-ide` (besucht am 09. 10. 2018).

[45] Sebastian Zarnekow. *Re: xText Standalone Setup Parsing a DSL from a File*. 3. Nov. 2010. URL: `https://www.eclipse.org/forums/index.php?t=msg&th=199632&goto=637050&#msg_637050` (besucht am 09. 10. 2018).

Bettini, in einem Beispielprojekt 100% Testabdeckung erreicht zu haben.[46] Als Nachweis dient ihm ein Screenshot der Eclipse-IDE. Das genutzte Tool scheint EclEmma gewesen zu sein.

Alle folgenden Betrachtungen sind nur auf den generierenden Code beschränkt. Es wurden in dieser Arbeit keine Tests für das Benutzerinterface geschrieben. Wichtig ist hier, zu entscheiden, was überhaupt getestet werden soll und über welchen Teilen des Codes die Abdeckungs-Metriken berechnet werden sollen. Schließt man etwa alles außer dem wirklich selbst entwickelten Code aus und berechnet die Metriken nur über den getesteten Klassen, so lässt sich eine vollständige Testabdeckung einfach erreichen, wie Abbildung 9 zeigt.

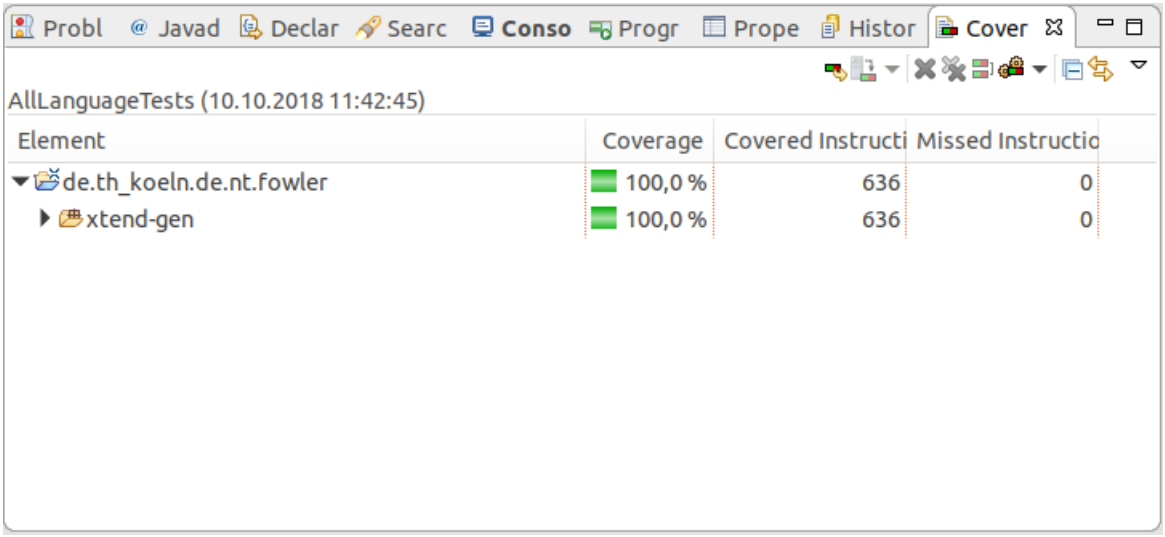


Abbildung 9: 100% Testabdeckung

Werden die Metriken bei gleichem Test auch über die Testklassen berechnet, sieht das Bild schon anders aus (Abbildung 10). Bei der Abdeckung der Testklassen fallen zwei Dinge auf.

Zum einen werden Anweisungen in den einzelnen Testklassen nicht ausgeführt. Das liegt daran, dass es sich um Java-Code handelt, der aus Xtend generiert wurde. Hier werden viele Testmethoden grundsätzlich von einem riesigen `try{}catch(Throwable _e){}`-Block umschlossen. Es wäre an dieser Stelle aber unsinnig, am Ende jedes Tests eine Exception zu werfen, nur um eine höhere Testabdeckung zu erreichen.

Zum andern handelt es sich um nicht ausgeführte Anweisungen in den Hamcrest-Matchern. Diese sollten selbstverständlich ordentlich getestet werden.

Aus Sicht des Autors sollten die Matcher aber in einer Bibliothek zusammengefasst und dort getestet werden, da hier im aktuellen Zustand eine Vermischung von Fachlichem Code und Test-Code vorliegt, die aufgelöst werden sollte.

[46] Bettini, *Testing Xtext Languages*.

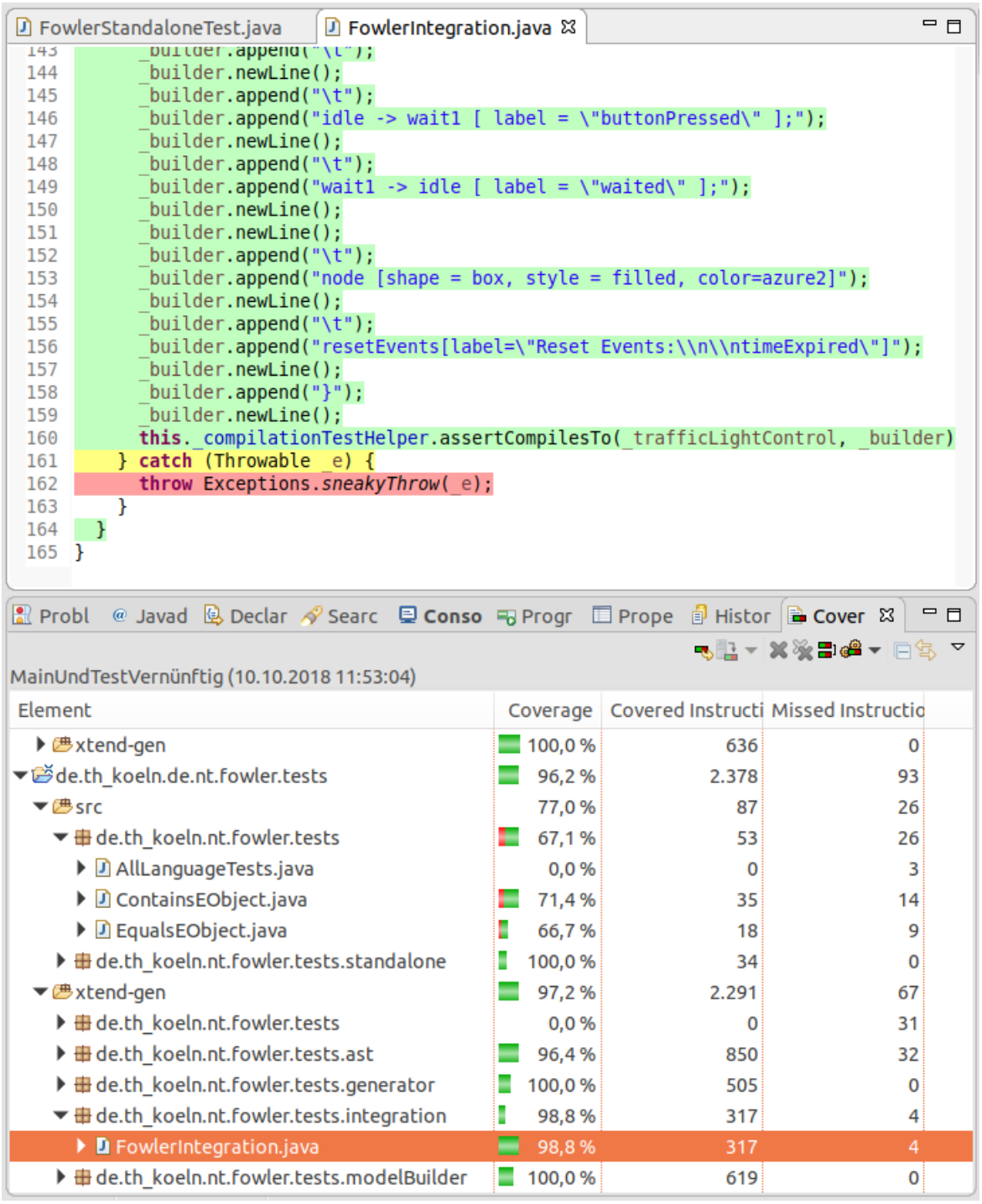


Abbildung 10: Testabdeckung inklusive Testklassen

Noch deutlicher wird die Situation, vor allem auch in absoluten Zahlen, wenn die Abdeckungs-Maße auch auf dem Bibliotheks-Code berechnet werden. Abbildung 11 zeigt deutlich, dass hier tausende Zeilen ungetesteter Code vorliegen. An dieser Stelle sollte aber klar gesagt werden, dass es unmöglich und unsinnig ist, allen genutzten Code aus Bibliotheken zu testen. Setzt man dem keine Grenzen, wäre es nötig, alle genutzten Bibliotheken der JVM, die JVM selber, das Betriebssystem, das BIOS oder EFI sowie den Microcode des Prozessors zu testen. Oder wie Ross

in der Einleitung zu seiner Dissertation schreibt: „It's no use, Mr. James – it's turtles all the way down.“[47]

Die Grenze kann vernünftigerweise nur so gezogen werden, dass der eigene Code möglichst gut getestet wird und Bibliotheken oder Frameworks sowie der von Frameworks wie *Antlr* generierte Code als korrekt angenommen wird. Bei der Auswahl von Bibliotheken muss aber eine sorgfältige Recherche zur Codequalität durchgeführt werden. Darauf, dass eine Bibliothek viel genutzt wird, kann man sich hier nicht verlassen. Ein Grund dafür könnte auch der Mangel an alternativen sein.

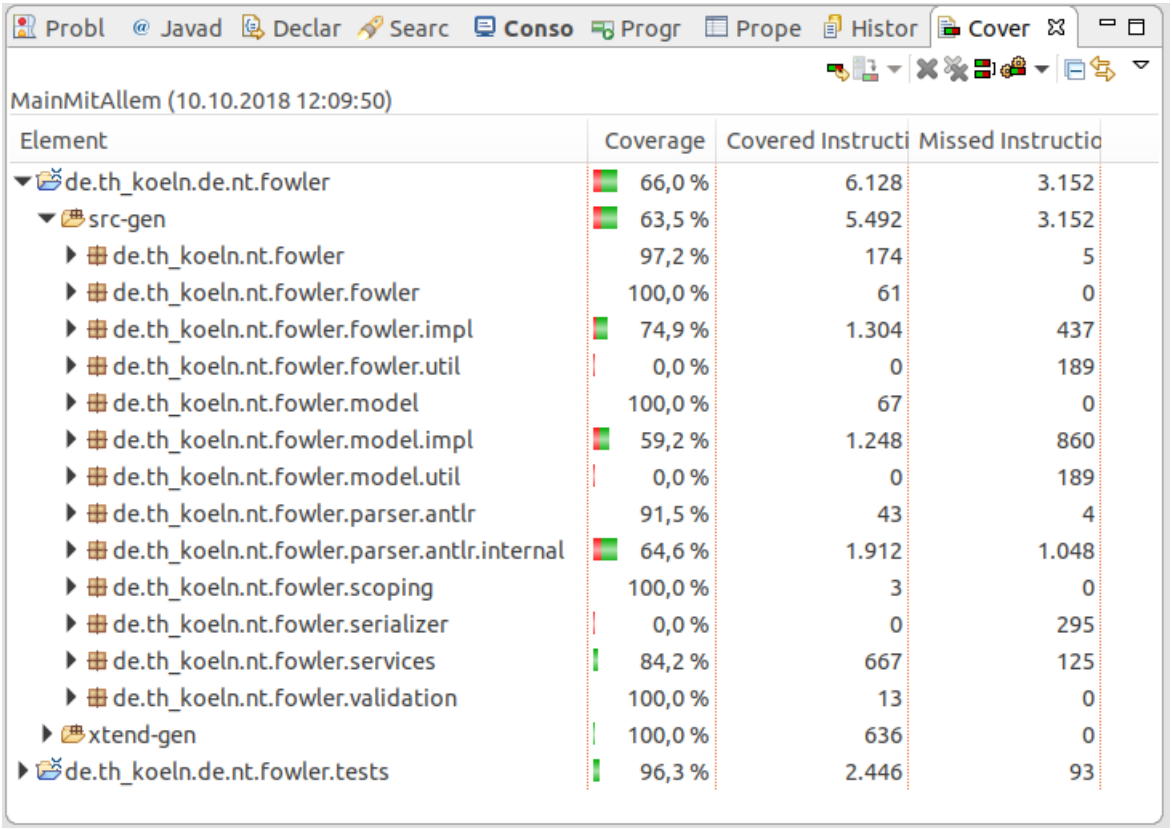
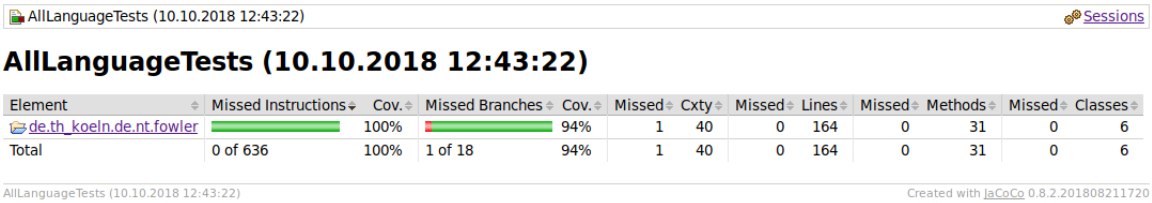


Abbildung 11: Testabdeckung inklusive Bibliotheken

Zuletzt stellt sich die Frage danach, welche Überdeckungsmaße überhaupt verwendet werden. EclEmma nutzt das Framework JaCoCo, um die Testabdeckung zu ermitteln. Exportiert man die sogenannte *Session* von JaCoCo als HTML-Report, kann man alle von JaCoCo berechneten Überdeckungsmaße sehen. Die in Eclipse angezeigten 100% Testabdeckung beziehen sich erst einmal nur auf das Maß *Anweisungsüberdeckung*.

[47] John Robert Ross. “Constraints on variables in syntax.” Diss. Massachusetts Institute of Technology, 1967. URL: `http://hdl.handle.net/1721.1/15166`, S. v.


AllLanguageTests (10.10.2018 12:43:22) Sessions

AllLanguageTests (10.10.2018 12:43:22)

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
de.th_koeln.de.nt.fowler		100%		94%	1	40	0	164	0	31	0	6
Total	0 of 636	100%	1 of 18	94%	1	40	0	164	0	31	0	6

AllLanguageTests (10.10.2018 12:43:22) Created with JaCoCo 0.8.2.201808211720

Abbildung 12: JaCoCo-Report zur Testabdeckung

Die Norm ISO/IED/IEEE 29119-4:2015[48] nennt folgende sechs Strukturbasierte Überdeckungsmaße:

- Statement Testing (Anweisungsüberdeckung, C0)
- Branch Testing (Zweigüberdeckung, C1)
- Decision Testing (Entscheidungsüberdeckung, C2)
- Branch Condition Testing (Bedingungsüberdeckung, C3)
- Branch Condition Combination Testing (Abdeckung aller Bedingungskombinationen, C4)
- Modified Condition Decision Coverage Testing

Die Grundlagen hierzu hat Myers schon 1979 gelegt.[49] Abbildung 12 zeigt, dass 100% Anweisungsüberdeckung nicht gleichbedeutend mit einem vollständigen Test ist. Im Gegenteil bezeichnet Myers dieses Maß als „so weak that it is generally considered to be useless“,[50] also so schwach, dass es generell unnütz ist. Welche Anforderungen tatsächlich an eine Software gestellt werden, ist immer eine Kosten-Nutzen-Abwägung und hängt im Einzelfall von der Anwendung der Software ab.

Sinnvoll ist sicherlich, eine Forderung für das Projekt aufzustellen, kritische Module aber mit einer höheren Anforderung zu belegen. Außerdem muss für Entscheidungsüberdeckung und aufwärts ein Werkzeug gefunden werden, welches die jeweilige Metrik auch misst.

Eine vollständige Pfadüberdeckung ist bei Software mit Schleifen nicht zu erreichen. Sogar wenn nur die Anforderung gestellt wird, Schleifen null mal, einmal, zweimal, eine typische und die maximale Anzahl mal zu durchlaufen, explodiert der Testaufwand für eine vollständige Pfadüberdeckung schnell und ist daher nicht praxisrelevant.

[48] ISO/IEC/IEEE. *ISO/IEC/IEEE 29119-4:2015(E): Software and systems engineering — Software testing — Part 4: Test techniques*. Dez. 2015.

[49] Glenford J. Myers. *The Art of Software Testing (Business Data Processing: A Wiley Series)*. Wiley, 1979, S. 36 ff.

[50] Myers, *The Art of Software Testing (Business Data Processing: A Wiley Series)*, S. 38.

4. Test Description Language

Wie schon in der Einleitung beschrieben, wurde die Test Description Language (TDL) im Rahmen eines Forschungsprojektes an der TH Köln implementiert. Ziel dabei ist es vor allem, für Hardware-Tests eine sogenannte „Single Source of Truth“ herzustellen. Dabei sollen die Stimuli der Hardware-Tests, das System zum Uberprüfen der Testergebnisse, die Konfiguration der Testhardware und die Vorlagen für die Dokumentation des Tests aus einer einzigen TDL-Datei generiert werden. Das führt, vergleichbar zu Software-Tests, auch im Hardware-Bereich zu wiederholbaren und, wenn entsprechendes Automated Test Equipment / Automatic Test Equipment (ATE) zur Verfügung steht, automatisierbaren Tests.

4.1. Die TDL in der Norm

Ursprünglich wurde die TDL als nicht unterteilter Standard veröffentlicht[51] und als solche auch von Lauber implementiert.[52] Während des Bearbeitungszeitraumes dieser Arbeit wurde dann vom ETSI eine neue, um sechs Teile erweiterte Version veröffentlicht. Diese enthält neben der schon 2014 veröffentlichten Abstrakten Syntax und der zugehörigen Semantik[53] auch eine graphische Syntax,[54] ein Austauschformat,[55] ein UML-Profil,,[56] eine Abbildung auf die Testing and Test Control Notation (TTCN-3)[57] sowie zwei Erweiterungen.[58,59] In der Zwischenzeit hat das Projekt eine eigene Webseite[60] und an der Universität Göttingen entsteht eine Referenzimplementierung der TDL.[61]

[51] ETSI, *ETSI ES 203 119: Methods for Testing and Specification (MTS); The Test Description Language (TDL); Specification of the Abstract Syntax and Associated Semantics*.

[52] Lauber, “Konzept und Entwicklung einer Testbeschreibungssprache und eines Systems zur Generierung von Testbenches und Testvektoren für einen Testautomaten”.

[53] ETSI, *ETSI ES 203 119-1: Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 1: Abstract Syntax and Associated Semantics*.

[54] ETSI. *ETSI ES 203 119-2: Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 2: Graphical Syntax*. Version 1.3.1. Mai 2018.

[55] ETSI. *ETSI ES 203 119-3: Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 3: Exchange Format*. Version 1.3.1. Mai 2018.

[56] ETSI. *ETSI ES 203 119-5: Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 5: UML profile for TDL*. Version 1.1.1. Mai 2018.

[57] ETSI. *ETSI ES 203 119-6: Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 6: Mapping to TTCN-3*. Version 1.1.1. Mai 2018.

[58] ETSI. *ETSI ES 203 119-4: Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 4: Structured Test Objective Specification (Extension)*. Version 1.4.1. Mai 2018.

[59] ETSI. *ETSI ES 203 119-7: Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 7: Extended Test Configurations*. Version 1.1.1. Mai 2018.

[60] ETSI. *Test Description Language*. 2018. URL: `https://tdl.etsi.org/`.

[61] Software Engineering For Distributed Systems Group. *TDL Reference Implementation (Phase 3) (STF 492)*. 2018. URL: `https://www.swe.informatik.uni-goettingen.de/research/tdl-reference-implementation-phase-3-stf-492` (besucht am 11. 10. 2018).

4.2. Abstrakte Syntax

Die Abstrakte Syntax, welche im ersten Teil der Norm standardisiert wird, ist in Form von MOF-konformen Diagrammen und als EBNF angegeben. Zusätzlich werden *Constrains* in Object Constraint Language (OCL) festgelegt. Als Beispiel, wie die Abstrakte Syntax in Xtext umgesetzt wurde, soll die `TestConfiguration` dienen.

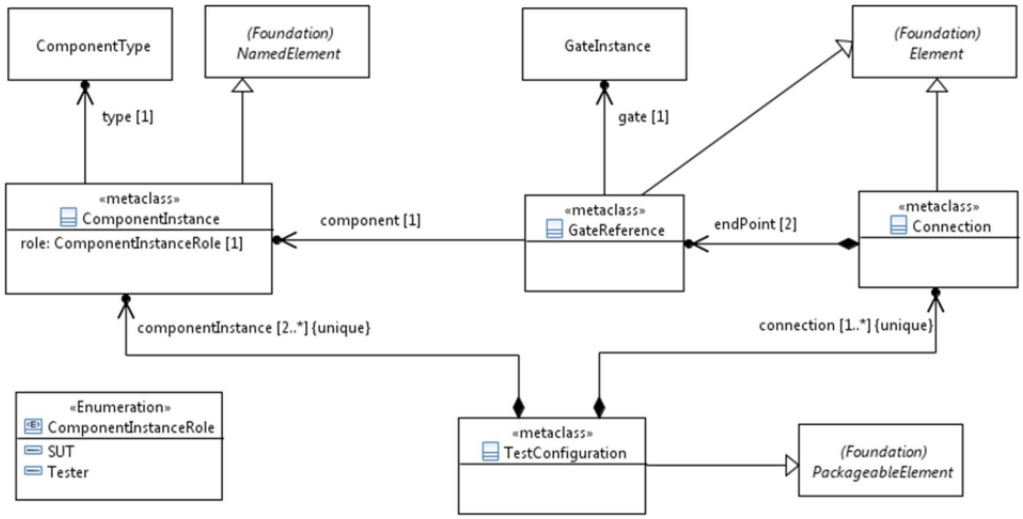


Abbildung 13: TestConfiguration Grafisch (ETSI ES 203 119-1 2018, S. 54)

Abbildung 13 zeigt die `TestConfiguration`, wie sie in der Norm dargestellt wird. Hier ist die Klasse selbst unten zentral abgeleitet von `PackageableElement`. Enthalten sind mindestens zwei Elemente vom Typ `ComponentInstance` und mindestens ein Element vom Typ `Connection`.

Die zugehörige Regel in der EBNF ist in Abbildung 14 zu sehen. Wichtig sind hier die Schlüsselwörter `Test Configuration`, mit welcher die Regel eingeleitet wird, der `EString`, die geschweiften Klammern und die Referenzen auf `ComponentInstance` und `Connection`.

```
TestConfiguration  ::=  'Test Configuration' EString '{' ComponentInstance {
                        ComponentInstance } Connection { Connection } '}' [ 'with' '{' [
                        Comment { Comment } ] [ Annotation { Annotation } ] '}' ] ;
```

Abbildung 14: TestConfiguration in EBNF (ETSI ES 203 119-1 2018, S. 109)

Der komplette optionale Block, der mit dem Schlüsselwort der `with` beginnt, kann an dieser Stelle ignoriert werden. Er ist an fast jeder Regel angehängt, um beliebigen Elementen einen Kommentar und eine Annotation anfügen zu können, wird aber weder in dieser Arbeit noch in der vorliegenden Implementierung verwendet.

4.3. Konkrete Syntax

Die Repräsentation der `TestConfiguration` in der konkreten Implementierung wird in Abbildung 15 gezeigt. Mit einigen Abweichungen entspricht diese der abstrakten Syntax.

```
206 TestConfiguration:
207     ('Test' 'Configuration' name=ID '{'
208         (COMPONENTINSTANCE+=ComponentInstance)+
209         (CONNECTION+=Connection)+
210         (ASSERTDEVIATION=AssertDeviation)
211         '}'
212         ('with' '{' (COMMENT+=Comment)* (ANNOTATION+=Annotation)* '}' )?
213     );
```

Abbildung 15: TestConfiguration in Xtext-EBNF (nach Lauber 2015, S. 52)

Die Abweichungen sind zum einen der zusätzliche Verweis auf eine `AssertDeviation`, zum anderen die Zuweisung aller variablen Elemente durch den „=“-Operator. Das sorgt dafür, dass entsprechende Felder oder Listen im AST erzeugt und die Elemente dort hinzugefügt werden.

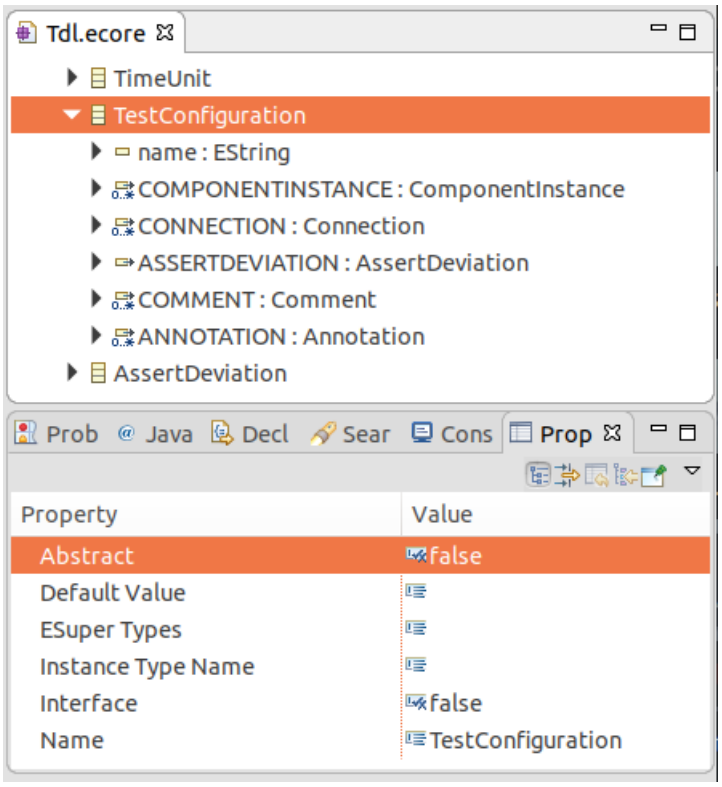


Abbildung 16: Modell der TestConfiguration im Ecore-Editor

Eine Besonderheit ist die Zuweisung `name=ID`. Damit wird dem besonderen Feld `name` ein String zugewiesen, dem durch das Terminalsymbol gewissen Einschränkungen auferlegt werden. So darf der Name nur Klein- und Großbuchstaben,

Ziffern und den Unterstrich enthalten, das erste Zeichen darf keine Ziffer sein und es darf ein Zirkumflex vorangestellt werden. Außerdem müssen alle Namen gleicher Elemente, das heißt in diesem Fall alle Namen von `TestConfigurations`, unterschiedlich und damit eindeutig sein. Wie Abbildung 16 zeigt, ist der Datentyp im generierten Ecore-Metamodell trotzdem korrekt `EString`.

4.4. Domänenmodell

Das Domänenmodell der TDL konnte im Bearbeitungszeitraum dieser Arbeit leider nicht in zufriedenstellender Qualität komplett erstellt werden. Trotz der eventuell noch nötigen Änderungen lassen sich doch einige Aussagen zur Benutzbarkeit treffen.

Während das Ecore-Modell des AST 81 Klassen enthält, sind es im Domänenmodell gerade einmal 15. Durch das deutlich kompaktere Modell lässt sich sehr gut damit arbeiten.

Listing 19 zeigt die Definition der `TestConfiguration`. Nach einer intensiven Auseinandersetzung mit der Norm fällt auf, dass die beiden Statements in den Zeilen 65 und 66 geändert werden müssen, sodass Listen vom Typ Design enthalten sind, da es von beiden `ComponentInstances` mehrere geben kann.

```
62 class TestConfiguration {
63    String name
64    String baseClock
65    contains Design dut
66    contains Design tester
67    contains AssertDeviation assertDeviation
68 }
```

Listing 19: Xcore-Modell der TDL

```
17  class ModelBuilderImpl implements ModelBuilder {
...
108     def createTestConfiguration(TestConfiguration testConfiguration) {
109         val de.th_koeln.nt.tdl.model.TestConfiguration tcToAdd =
            ↪  modelFactory.createTestConfiguration
110         // Search for the DuT object in the AST part:
111         val ComponentInstance dut =
            ↪  testConfiguration.COMPONENTINSTANCE.filter [
112             it.COMPONENTINSTANCEROLE == ComponentInstanceRole.DUT
113         ].head
114
115         tcToAdd.name = testConfiguration.name
116         //tcToAdd.baseClock = ; // TODO
117         tcToAdd.dut = createDut(dut);
118         model.testConfigurations.add(tcToAdd)
119     }
```

Listing 20: Ausschnitt aus dem ModelBuilder

4.5. ModelBuilder

Der ModelBuilder ist ähnlich aufgebaut wie derjenige aus Abschnitt 3.7.2. Um den Code gut testen zu können, müssen einige Dinge beachtet werden.

Lange Methoden, die komplizierte Abläufe enthalten, sollten, wo immer sinnvoll möglich, aufgeteilt werden. Komplizierte Filter können in eigene Methoden ausgelagert werden, um unendlich lange, verschachtelte Zeilen zu verhindern. Besonders wichtig bei großen ASTs ist es, den ModelBuilder so zu schreiben, dass er mit unvollständigen Syntaxbäumen zurecht kommt.

Ein Beispiel dazu ist der Code in Listing 20. Dort wird in den Zeilen 111 bis 113 das erste DUT gesucht. Neben der in Abschnitt 4.4 angesprochenen Problematik, dass es mehrere DUTs geben könnte, birgt dieser Code eine zusätzliche Falle. Wenn in einem Test kein Code getestet werden soll, der ein DUT betrifft, sondern ausschließlich solcher, der eine `ComponentInstance` vom Typ `Tester` enthält, so wird in der Testklasse ein AST erstellt, in dem eine `TestConfiguration` mit einem `Tester` vorhanden ist. Die Zuweisung in Zeile 111 wird dann der Variablen `dut` null zuweisen (eine Nullreferenz, nicht die Zahl 0) und diese dann in Zeile 117 an die Methode `createDut(Design dut)` übergeben. Wird dort nicht überprüft, dass das übergebene DUT nicht null ist, wird der Test mit einer `NullPointerException` beendet.

4.6. Code-Generatoren

Besonders interessant ist ein Verlgeich der Code-Generatoren. Allein die Methode, welche über alle `TestDescriptions` iteriert und für jede einen passenden Ordner mit allen zugehörigen Dateien erzeugt, wurde von 16 Zeilen auf 7 Zeilen reduziert (siehe Listings 21 und 22). Diese besteht auch nicht mehr aus fünf ineinander verschachtelten Schleifen sondern nur noch semantisch korrekt aus einer.

```
override void doGenerate(Resource resource, IFileSystemAccess fsa) {
   for (tdlsp :
    ↪  resource.allContents.toIterable.filter(typeof(TDLSpec))) {
      for (tc :
       ↪  resource.allContents.toIterable.filter(typeof(TestConfiguration)))
       ↪  {
         for (sac : resource.allContents.toIterable.filter(
          ↪  typeof(SignalAdapterConfiguration))) {
            for (td : resource.allContents.toIterable.filter(
             ↪  typeof(TestDescription))) {
               if (sac !== null) {
                  for (compinst : tc.COMPONENTINSTANCE) {
                     val folderName = tdlsp.name + "/" + tc.name +
                      ↪  "/" + td.name + "/"
                     fsa.generateFile(folderName + "config.yaml",
                      ↪  generate_config(tc, sac, td))
                  }
               }
            }
         }
      }
   }
}
```

Listing 21: Compiler-Schleife ohne Domänenmodell

```
override void doGenerate(Resource resource, IFileSystemAccess2 fsa,
 ↪  IGeneratorContext context) {
   modelBuilder.populateDomainModel(resource)
   modelBuilder.domainModel.tests.forEach[
      val folderName = modelBuilder.domainModel.name + "/" +
       ↪  it.testConfiguration.name + "/" + it.name + "/"
      fsa.generateFile(folderName + "config.yaml",
       ↪  ConfigFileGenerator.generate(modelBuilder.domainModel, it))
   ]
}
```

Listing 22: Compiler-Schleife mit Domänenmodell

Ein Vergleich der beiden eigentlichen Generatoren für die Konfigurationsdateien zeigt, dass sich die Struktur nicht stark geändert hat. Es ist aber im direkten Vergleich deutlich weniger zusätzlicher Code nötig. Bei der Entwicklung von Codegeneratoren fällt auf, dass immer wieder zusätzliche Zähler und Listen benötigt werden. Die Anzahl der Zähler konnte von fünf auf einen reduziert werden, auf zusätzliche Indizes konnte komplett verzichtet werden (siehe Listings 23 und 24).

```
137     def generate_config(TestConfiguration tc,
        ↪  SignalAdapterConfiguration sac, TestDescription td) {
138        var LogicLevel[] knownLogicLevel = knownLogicLevel()
139        var LogicLevel[] useLogicLevel =
           ↪  sac.usedLogicLevel(knownLogicLevel)
140        val symbolNameList = newArrayList("!", "-", "+", "%", "&", "/",
           ↪  "(", ")", "=", "?")
141        var j = 0
142        var index = -1
143        var inputSignalAdapter = 0
144        var outputSignalAdapter = 0
145        var int[] alrdyUsed = newIntArrayOfSize(useLogicLevel.length)
146
147        for (i : 0 ..< alrdyUsed.length) {
148           alrdyUsed.set(i, 0) // init
149        }
150        var counter = 0;
```

Listing 23: Kopf des ConfigFileGenerators ohne Domänenmodell

```
8      def static String generate(TDLSpec spec, TestInstance
       ↪  testInstance) {
9         var signalAdapterCounter = 0
```

Listing 24: Kopf des ConfigFileGenerators mit Domänenmodel

Im eigentlichen Generator-Code werden durch das kompaktere Modell Datenzugriffe deutlich verkürzt. Der Zugriff auf die Seriennummer eines Signaladapters erfolgt ohne Domänenmodell durch `sa.SERIAL_NUMBER.name`, mit Domänenmodell durch `sa.serialNumber`. Um den Programmcode ohne Domänenmodell halbwegs übersichtlich zu halten wurden vier sogenannte *Extension Classes* benutzt, mit denen man in Xtend das Interface einer bestehenden Klasse erweitern kann. Das zusammen sind 182 Zeilen Programmcode - mehr als der Teil des `ModelBuilders`, welcher für den `ConfigFileGenerator` benötigt wird.

4.7. Testen der TDL

Das Testen der TDL entspricht prinzipiell dem Vorgehen in Abschnitt 3.9. Dadurch, dass so gut wie jedes Element als `PackageableElement` definiert wird, ist das Aufbauen eines AST jedoch ungleich schwerer. Anstatt zum Beispiel eine `TestConfiguration` einfach der `TDLSpec` hinzuzufügen, muss diese vorher, wie in Zeile 43 von Listing 25, in ein `PackageableElement` „verpackt“ werden.

Leider schlägt der in Listing 25 gezeigte Test fehl, da der ModelBuilder nicht, wie in Abschnitt 4.5 gefordert gegen Null-Pointer gesichert wurde. Obwohl, bedingt durch die auch mit Domänenmodell immer noch große Komplexität der Sprache, für die TDL keine befriedigende Testbasis mehr entwickelt wurde, konnte doch anhand der einfacheren Sprache für Zustandsautomaten gezeigt werden, dass durch die Einführung eines Domänenmodells die Kopplung der Codegeneratoren an die Sprachsyntax eliminiert werden kann. Der Testaufwand wird zwar etwas größer, die Tests lassen sich jedoch deutlich besser formulieren.

```
25  class TdlModelBuilderTest {
...
38      @Test
39      def void eventsArePopulatedIntoModel() {
40          val TDLSpec spec = astFactory.createTDLSpec => [name = "de0t"]
41          val TestConfiguration tc = astFactory.createTestConfiguration
            ↪  => [name = "de0t_cf"]
42
43          spec.PACKAGEABLEELEMENT.add(astFactory.createPackageableElement
            ↪  => [TESTCONFIGURATION = tc]);
44
45          ast.contents.add(spec)
46
47          modelBuilder.populateDomainModel(ast)
48
49          assertThat("TDLSpec is not correctly added",
50              modelBuilder.domainModel.eAllContents,
51              containsEObject(modelFactory.createTDLSpec => [it.name =
                ↪  "de0t"]))
52          assertThat("TestConfiguration is not created",
53              modelBuilder.domainModel.eAllContents,
54              containsEObject(modelFactory.createTestConfiguration =>
                ↪  [name = "de0t_cf"]))
55      }
```

Listing 25: TdlModelBuilderTest.xtend

5. Bewertung

Nachdem an einer einfachen Sprache für Zustandsautomaten und der deutlich komplexeren TDL gezeigt wurde, wie sich ein echtes Domänenmodell in eine Xtext-Sprache einfügt, soll abschließend bewertet werden, ob sich der zusätzliche Aufwand lohnt. Testen in der Software-Entwicklung ist immer eine Kosten-Nutzen-Rechnung. Ein zusätzlicher Testaufwand muss also einen Gewinn bringen.

5.1. Bewertung des Xcore-Modells

Wird allein mit Eclipse entwickelt, fügt sich ein Xcore-Modell nahtlos in den Build ein. Es reicht, eine Xcore-Datei in den Ordner `model` zu legen. Dadurch werden automatisch die nötigen Anpassungen der Konfiguration vorgenommen und die Java-Klassen des Modells werden automatisch in den auch von Xtext genutzten Ordner `src-gen` generiert.

Soll aber Maven als Build-System zum Einsatz kommen, muss über eine Alternative nachgedacht werden. Trotz intensiver Recherche und einiger Versuche konnte Maven im Bearbeitungszeitraum dieser Arbeit nicht so konfiguriert werden, dass es sowohl den für Xtext nötigen MWE2-Build als auch das Xcore-Modell kompiliert hätte. Auch in den MWE2-Build von Xtext konnte das zusätzliche Modell nicht so eingefügt werden, dass die Java-Klassen auch generiert würden.

Im Hinblick auf die Ausdrucksstärke der Xcore-Sprache zur Modellierung von Datenmodellen und die vorgesehene Option, ein Xcore-Modell für den AST einer Xtext-Sprache zu nutzen, ist das sowohl schade als auch verwunderlich. Dies ist die persönliche Meinung des Autors und nicht objektiv.

5.2. Bewertung des ModelBuilders

Der `ModelBuilder` lässt sich bei einfachen Sprachen sehr gut und bei komplexeren Sprachen gut umsetzen. Es wurden bei der Entwicklung einige Muster festgestellt, die sich wiederholen. Das ist zum einen der einfache Datentransport vom AST ins Domänenmodell, zum anderen die Strukturänderung zwischen diesen. Während ersterer sicherlich durch ein Framework zum Model-Mapping einfacher gestaltet werden könnte (am besten mit einer eigenen DSL), dürfte letztere damit schwieriger zu handhaben sein. Leider konnte aus Zeitmangel kein sauberes Muster zum Umgang mit unvollständigen ASTs erstellt werden. Wenn dieses Problem ordentlich gelöst wird, ist ein sauber modularisierter `ModelBuilder` ein wirklich gutes Muster.

5.3. Bewertung der Test- und Wartbarkeit im Xtext-Umfeld

Bettini hat sehr gut ausgearbeitet, wie Xtext-Sprachen mit den Sprachfunktionen von Xtend gut getestet werden können und wie diese Tests lesbar bleiben.[62] Die Arbeit mit AST und Domänenmodell ist deutlich übersichtlicher als dies in Java der Fall wäre, wodurch die Tests an Ausdrucksstärke gewinnen.

Durch die Einführung eines Domänenmodells werden die Codegeneratoren von der Sprache selber entkoppelt. Die Codegeneratoren können mit dem Domänenmodell deutlich einfacher gestaltet werden, da dieses strukturell dem generierten Code deutlich näher ist als der AST. Die einzelnen AST-Elemente werden an deutlich weniger Stellen im Code genutzt (vor allem nur noch im `ModelBuilder`), sodass Änderungen an der Sprachsyntax einfacher werden. Der Nutzen des Domänenmodells ist an dieser Stelle am deutlichsten erkennbar. Da zumindest alle Strukturänderungen auch in einem Codegenerator ohne Domänenmodell stattfinden müssen, erhöht sich der Entwicklungsaufwand auch nicht so stark wie zuerst angenommen.

Die in Abschnitt 3.9.1 Testvorbereitung vorgestellten Hamcrest-Matcher stellen grundlegende Funktionalität zur Verfügung. Wünschenswert wäre eine etwas umfangreichere Bibliothek an Matchern, die nicht nur auf dem `TreeIterator` arbeiten, sondern auf der Baumstruktur des Abstract Syntax Tree. Hier wäre ein Framework von Hamcrest-Matchern mit einer Strukturerkennung sehr sinnvoll. Dazu müssten Hamcrest-Matcher zur Navigation in der Baumstruktur definiert werden, die sowohl gut lesbar als auch typsicher ist.

Im Hinblick auf die Probleme, denen die Entwicklung von Hamcrest momentan unterworfen ist,[63] muss allerdings die Frage gestellt werden, ob Hamcrest nicht zu Gunsten von AssertJ verworfen werden sollte. Bei der Benutzung von Hamcrest sind auch massive infrastrukturelle Probleme aufgetreten. Zum einen bindet Junit4 das Paket hamcrest-core ein. Möchte man die dort nicht vorhandenen Matcher nutzen, muss man hamcrest-all zusätzlich einbinden. Dieses muss allerdings beim Build *vor* Junit4 im Classpath liegen, da es ansonsten zu Problemen mit unterschiedlich signiertem Code kommt.

Das andere Problem ist das Einbinden von Hamcrest in den Maven-Build. Das liegt an einer komplexen Kombination von Buildphasen im Maven-Build mit falscher Einordnung des Tycho-Plugins, welches zum Kompilieren von Eclipse-Plugins genutzt wird und der Weigerung des Tycho-Compilers, Hamcrest oder andere Abhängigkeiten aus dem Maven-Repository zu benutzen.

[62] Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend*, S. 123 ff.

[63] Zac Sweers u. a. *Is this still being maintained?* 23. Dez. 2015. URL: `https://github.com/hamcrest/JavaHamcrest/issues/131` (besucht am 02. 10. 2018).

5.4. Einordnung in einen Testprozess

Die Normen zur Funktionalen Sicherheit schreiben vor, dass die Entwicklung von Hard- und Software in einen Prozess eingeordnet wird. Dafür bieten sie Vorlagen an, es ist aber weniger wichtig, diese exakt umzusetzen, als den Entwicklungsprozess zu dokumentieren, einzuhalten und auch die Einhaltung zu dokumentieren. Das gilt natürlich auch für das Testen von Software. Auch bei der relativ neuen Norm zum Testen von Software, ISO 29119, folgt auf den ersten Teil „Concepts and definitions“ nicht eine Beschreibung, was und wie getestet werden soll, sondern „Test Processes“.[64]

Damit ist allerdings kein technischer Prozess wie Continuous Integration gemeint. Vielmehr geht es darum, wie das Testen organisatorisch behandelt wird, das heißt, welche Vorbereitungen getroffen werden, welche Abläufe eingehalten werden und welche Dokumente angefertigt werden müssen. Auch geregelt wird, wie dieser Prozess im Entwicklungsteam und der gesamten Organisation durch- und umgesetzt wird.

An dieser Stelle bleibt leider die Frage offen, wie ein solcher Prozess in einer Hochschule mit studentischen Abschlussarbeiten, wechselnden Mitarbeitern und Verantwortlichkeiten umgesetzt werden könnte. Für einzelne Projekte ist die Umsetzung sicherlich möglich. Ob es jedoch auch möglich ist, einen solchen Prozess in einem Institut oder gar einer kompletten Fakultät umzusetzen, müsste untersucht werden. Denn auch an dieser Stelle dürfen die Kosten den Nutzen nicht übersteigen.

[64] ISO/IEC/IEEE. *ISO/IEC/IEEE 29119-2:2015(E): Software and systems engineering — Software testing — Part 2: Test processes.* 1. Sep. 2013.

6. Fazit

In der professionellen Softwareentwicklung wird schon lange nicht mehr bezweifelt, dass Testen grundsätzlich erforderlich ist. Dies gilt besonders für Anwendungen, die sicherheitskritisch sind, soll damit doch das Risiko von Gefahren für Mensch und Umwelt in einem Bereich gehalten werden, der zu verantworten ist. Das gilt auch für Tools, mit denen solche Anwendungen erstellt werden.

Tests dienen nicht nur dazu zu überprüfen, ob die Spezifikation eingehalten wird. Sie sollen auch sicherstellen, dass bei Änderungen oder dem Beseitigen von Fehlern keine neuen Fehler auftreten. Nur mit ausreichenden und aussagekräftigen Tests lassen sich diese Ziele erreichen.

Es konnte in dieser Arbeit an zwei unterschiedlich komplexen Beispielen gezeigt werden, dass ein Domänenmodell dazu beiträgt, bei komplexen Xtext-Sprachen besser testen zu können. Gleichzeitig entkoppelt es Änderungen an der Sprachsyntax von den Codegeneratoren und umgekehrt. Aber auch auf den erhöhten Aufwand durch den Teil der Software, welcher den Abstract Syntax Tree (AST) in das Domänenmodell umformt, wurde aufmerksam gemacht. In der Praxis bleibt immer abzuwägen, ob der Nutzen diesen zusätzlichen Aufwand übersteigt. Die einfache Sprache für Zustandsautomaten lässt sich auch ohne Domänenmodell nutzen, die Test Description Language (TDL) profitiert deutlich davon. Die Grenze liegt irgendwo dazwischen und hängt wohl auch von der Anwendung der Sprache ab.

Auch die Forderungen, welche an die Testabdeckung gestellt werden, müssen in diese Abwägung einfließen. Dieses Thema wurde bezüglich der strukturell orientierten Tests (Whitebox-Tests) ausgeführt, muss aber auch auf Schnittstellen-Tests (Blackbox-Tests), Integrationstests und Akzeptanztests ausgeweitet werden.

6.1. Herausforderungen

Auf die Problematik, dass sich Xcore und Hamcrest nicht sauber in den Maven-Build eingliedern lassen, wurde schon hingewiesen. Das Xcore-Modell könnte in ein eigenes Eclipse-Projekt ausgelagert werden, denn grundsätzlich ist der Build mit Maven wohl möglich. Allerdings muss dieses Projekt dann in den Eclipse-Build eingebunden werden.

Auch die in den vergangenen Jahren nur sehr schleppend vorankommende Entwicklung von Hamcrest wurde angedeutet. Im Anbetracht der Probleme, die es beim Einbinden in den Build macht, sollte geprüft werden, ob ein Umstieg auf *AssertJ* nicht sinnvoll ist.

6.2. Ausblick

Wenn in einem Projekt erfolgreich ein Domänenmodell etabliert werden konnte, bietet es sich an, dieses auch in anderen Tools zu nutzen. Im Umfeld der TDL-Implementierung sind einige kleinere Programme entstanden, die ihre Daten aus der Konfigurationsdatei entnehmen. Da diese fast dieselbe Struktur wie die Konfiguration im Domänenmodell hat, wäre es einfach, dieses auch in den anderen Tools einzusetzen. Dazu müsste die Modell-Definition unabhängig von der Xtext-Sprache entwickelt und gepflegt werden.

Wird dieser Weg eingeschlagen, wäre es sinnvoll, die konkreten, aus einem TDL-Skript abgeleiteten, Domänenmodell-Instanzen abzuspeichern um sie in den anderen Tools laden zu können. Ecore bietet mit den Methoden `Resource.load()` und `Resource.save()` dazu die Möglichkeit. Diese sind auch nicht auf Dateien beschränkt, sondern können dazu `InputStream` respektive `OutputStream` nutzen. Damit wäre auch eine direkte Ablage in einer Datenbank denkbar.

Im Zuge der Recherche ist aufgefallen, dass das ETSI ein Ecore-Modell der TDL, wie sie normiert wurde, entwickelt hat und zur Verfügung stellt. Dieses gleicht strukturell dem AST und nicht dem in dieser Arbeit entwickelten Domänenmodell. Trotzdem könnte es nützlich sein, dieses zu importieren und in der Xtext-Syntax zu verwenden. Die eigenen Erweiterungen müssten dann als zusätzliches Modell gepflegt werden. Damit würde aber auch deutlicher, welcher Teil der Eigenentwicklung standardkonform ist und welcher nicht, und es wäre einfacher, einen möglichst großen Teil der Sprache konform zu halten.

Laut der TDL-Webseite versteht das ETSI die TDL als Aufsatz auf die Sprache Testing and Test Control Notation (TTCN-3).[65] Dort wurde auch erkannt, dass in der TDL eine Konfiguration der Testhardware nicht möglich und auch nicht vorgesehen ist. Die Konfiguration soll unabhängig vom Test gehalten werden. Ob das dort erwähnte Vorgehen im Zusammenspiel mit dem in Abschnitt 2.2 vorgestellten Testautomaten möglich und sinnvoll ist, müsste geprüft werden.

Auch möglich wäre eine Kooperation mit dem Institut für Informatik an der Universität Göttingen. Dort entsteht die Referenz-Implementierung der TDL und der Standard wird dort auch mitentwickelt.

[65] ETSI, *Test Description Language*.

Literatur

Bettini, Lorenzo. *Implementing Domain-Specific Languages with Xtext and Xtend*. 2nd Revised edition. Birmingham: Packt Publishing, 31. Aug. 2016.

– *Testing Xtext Languages*. ab 27;58. 9. Nov. 2015. URL: `https://youtu.be/fTkwFJy3EeM?t=1678` (besucht am 07. 10. 2018).

Corporation, IBM u. a. *Interface EObject*. 2006. URL: `http://download.eclipse.org/modeling/emf/emf/javadoc/2.4.2/org/eclipse/emf/ecore/EObject.html` (besucht am 07. 10. 2018).

DIN. *DIN EN 50128; VDE 0831-128:2012–03: Bahnanwendungen - Telekommunikationstechnik, Signaltechnik und Datenverarbeitungssysteme - Software für Eisenbahnsteuerungs- und Überwachungssysteme; Deutsche Fassung EN 50128:2011*. März 2012.

Efftinge, Sven. *How And Why Use Xtext Without The IDE*. 16. März 2016. URL: `https://typefox.io/how-and-why-use-xtext-without-the-ide` (besucht am 09. 10. 2018).

– *initial commit*. 9. Mai 2008. URL: `https://github.com/eclipse/xtext/commit/e6d6a78063ae6ea5d80b25ace7b4416b0e169654` (besucht am 10. 10. 2018).

ETSI. *ETSI ES 203 119-1: Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 1: Abstract Syntax and Associated Semantics*. Version 1.4.1. Mai 2018.

– *ETSI ES 203 119-2: Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 2: Graphical Syntax*. Version 1.3.1. Mai 2018.

– *ETSI ES 203 119-3: Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 3: Exchange Format*. Version 1.3.1. Mai 2018.

– *ETSI ES 203 119-4: Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 4: Structured Test Objective Specification (Extension)*. Version 1.4.1. Mai 2018.

– *ETSI ES 203 119-5: Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 5: UML profile for TDL*. Version 1.1.1. Mai 2018.

– *ETSI ES 203 119-6: Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 6: Mapping to TTCN-3*. Version 1.1.1. Mai 2018.

– *ETSI ES 203 119-7: Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 7: Extended Test Configurations*. Version 1.1.1. Mai 2018.

ETSI. *ETSI ES 203 119: Methods for Testing and Specification (MTS); The Test Description Language (TDL); Specification of the Abstract Syntax and Associated Semantics*. Version 1.1.1. Apr. 2014.

– *Test Description Language*. 2018. URL: https://tdl.etsi.org/.

Evans, Eric. *Domain-Driven Design Reference. Definitions and Pattern Summaries*. März 2015. URL: https://domainlanguage.com/wp-content/uploads/2016/05/DDD_Reference_2015-03.pdf.

Foundation, Eclipse. *Eclipse Modeling Framework (EMF)*. 2018. URL: https://www.eclipse.org/modeling/emf/ (besucht am 10. 10. 2018).

Foundation, The Apache Software. *What is Velocity?* 2016. URL: https://velocity.apache.org/ (besucht am 10. 10. 2018).

Fowler, Martin. *Domain-Specific Languages*. Addison-Wesley Signature Series. Boston: Pearson Education, 2010.

– *FluentInterface*. 20. Dez. 2005. URL: https://martinfowler.com/bliki/FluentInterface.html.

Group, Object Management. *OMG Meta Object Facility (MOF) Core Specification*. Nov. 2016. URL: https://www.omg.org/spec/MOF (besucht am 11. 10. 2018).

Group, Software Engineering For Distributed Systems. *TDL Reference Implementation (Phase 3) (STF 492)*. 2018. URL: https://www.swe.informatik.uni-goettingen.de/research/tdl-reference-implementation-phase-3-stf-492 (besucht am 11. 10. 2018).

IEC. *IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems*. Apr. 2010.

ISO. *ISO 26262: Road vehicles – Functional safety*. Nov. 2011.

ISO/IEC/IEEE. *ISO/IEC/IEEE 29119-2:2015(E): Software and systems engineering — Software testing — Part 2: Test processes*. 1. Sep. 2013.

– *ISO/IEC/IEEE 29119-4:2015(E): Software and systems engineering — Software testing — Part 4: Test techniques*. Dez. 2015.

Klaus Kreft, Angelika Langer. *Hash-Code-Berechnung Wie, wann und warum implementiert man die hashCode()-Methode? Teil 1: Die Prinzipien der Implementierung von equals()*. Jan. 2002. URL: http://www.angelikalanger.com/Articles/EffectiveJava/03.HashCode/03.HashCode.html (besucht am 07. 10. 2018).

– *Objektvergleich Wie, wann und warum implementiert man die equals()-Methode? Teil 1: Die Prinzipien der Implementierung von equals()*. Jan. 2002. URL: http://www.angelikalanger.com/Articles/EffectiveJava/01.Equals-Part1/01.Equals1.html (besucht am 07. 10. 2018).

– *Objektvergleich Wie, wann und warum implementiert man die equals()-Methode? Teil 2: Der Vergleichbarkeitstest*. März 2002. URL: http://www.angelikalanger.

com/Articles/EffectiveJava/02.Equals-Part2/02.Equals2.html (besucht am 07. 10. 2018).

Krawutschke, Tobias u. a. "Test automation for reengineered modules using test description language and FPGA". In: *embedded world Conference 2018 – Proceedings*. embedded world Conference 2018 – Proceedings: WEKA FACHMEDIEN GmbH, 2018.

Lauber, David. "Konzept und Entwicklung einer Testbeschreibungssprache und eines Systems zur Generierung von Testbenches und Testvektoren für einen Testautomaten". Masterarbeit. Fachhochschule Köln, 18. Juni 2015.

Lüth, René. "Erzeugung von Testfällen mit einem DSL-basierten Generator auf Basis der ETSI-TDL für einen digitalen Testautomaten". Bachelorarbeit. TH Köln, 22. Feb. 2017.

Myers, Glenford J. *The Art of Software Testing (Business Data Processing: A Wiley Series)*. Wiley, 1979.

o.V. *Maven: The Complete Reference*. o.J. URL: http://books.sonatype.com/mvnref-book/pdf/mvnref-pdf.pdf (besucht am 02. 05. 2018).

– *Webseite: Model-to-Model Transformation in Eclipse*. o.J. URL: https://projects.eclipse.org/projects/modeling.mmt (besucht am 20. 09. 2018).

– *Xtend Documentation*. o.J. URL: https://www.eclipse.org/xtend/documentation/101_gettingstarted.html.

– *Xtext Dokumentation*. o.J. URL: https://www.eclipse.org/Xtext/documentation/index.html (besucht am 03. 08. 2018).

Parr, Terence. *Language implementation patterns : create your own domain-specific and general programming languages*. Raleigh, N.C: Pragmatic Bookshelf, 2010.

– *The Definitive ANTLR Reference*. English. Raleigh, NC: Pragmatic Bookshelf, 2007.

Ross, John Robert. "Constraints on variables in syntax." Diss. Massachusetts Institute of Technology, 1967. URL: http://hdl.handle.net/1721.1/15166.

Schulze, Martin. "Defining a Domain Specific Language for the Configuration of SoPCs Based on the OpenRISC Platform". Bachelorarbeit. Fachhochschule Köln, Juli 2013.

Spiller, Martin. *Maven 3: Konfigurationsmanagement mit Java*. 1. Aufl. Verfasserangabe: Martin Spiller ; Quelldatenbank: FHBK-x ; Format:marcform: print ; Umfang: 356 S. : Ill. ; 978-3-8266-9118-8 Pb. : EUR 29.95 (DE); 3-8266-9118-0 Pb. : ca. EUR 29.95 (DE). Heidelberg u.a.: Mitp, 2011. URL: http://digitale-objekte.hbz-nrw.de/storage/2011/09/24/file%5C_12/4297430.pdf.

Sweers, Zac u. a. *Is this still being maintained?* 23. Dez. 2015. URL: https://github.com/hamcrest/JavaHamcrest/issues/131 (besucht am 02. 10. 2018).

Vogel, Lars. *Testing with AssertJ assertions - Tutorial*. 29. Aug. 2016. URL: `http://www.vogella.com/tutorials/AssertJ/article.html` (besucht am 07. 10. 2018).

Wegner, Miro. *A Review of Java Template Engines*. 2. Dez. 2016. URL: `https://dzone.com/articles/template-engines-at-one-spring-boot-and-engines-se` (besucht am 10. 10. 2018).

Zarnekow, Sebastian. *Re: xText Standalone Setup Parsing a DSL from a File*. 3. Nov. 2010. URL: `https://www.eclipse.org/forums/index.php?t=msg&th=199632&goto=637050&#msg_637050` (besucht am 09. 10. 2018).

A. Komplette Listings

A.1. Kapitel 3: DSL für Endliche Automaten

```
1  package de.th_koeln.nt.fowler.generator
2
3  import org.eclipse.emf.ecore.resource.Resource
4  import org.eclipse.xtext.generator.AbstractGenerator
5  import org.eclipse.xtext.generator.IFileSystemAccess2
6  import org.eclipse.xtext.generator.IGeneratorContext
7  import de.th_koeln.nt.fowler.fowler.State
8  import de.th_koeln.nt.fowler.fowler.Statemachine
9
10 class FowlerGenerator extends AbstractGenerator {
11
12    override void doGenerate(Resource resource, IFileSystemAccess2 fsa,
   ↪  IGeneratorContext context) {
13       fsa.generateFile("fsm.dot", graphviz(resource))
14    }
15
16    def String graphviz(Resource resource) {
17       '''
18       digraph finite_state_machine {
19           size="8,5"
20
21
22
23           node [shape = record, style = rounded];
24          «FOR state :
   ↪  resource.allContents.toIterable.filter(typeof(State))»
25           «state.name»[label="{«state.name»|«FOR action :
   ↪  state.actions»«action.code» «ENDFOR»}"];
26           «ENDFOR»
27
28           «FOR state :
   ↪  resource.allContents.toIterable.filter(typeof(State))»
29           «FOR transition : state.transitions»
30           «state.name» -> «transition.state.name» [ label =
   ↪  "«transition.event.name»" ];
31            «ENDFOR»
32            «ENDFOR»
33
34         node [shape = box, style = filled, color=azure2]
35         resetEvents[label="Reset Events:\n«FOR resetEvent :
   ↪  resource.allContents.toIterable
36           .filter(typeof(Statemachine))
37           .last.resetEvents»\n«resetEvent.name»«ENDFOR»"]
38
39       }
40       '''
41    }
42 }
```

Listing 26: Graphviz-Generator, erste Erweiterung

```
1  package de.th_koeln.nt.fowler.generator
2
3  import org.eclipse.emf.ecore.resource.Resource
4  import org.eclipse.xtext.generator.AbstractGenerator
5  import org.eclipse.xtext.generator.IFileSystemAccess2
6  import org.eclipse.xtext.generator.IGeneratorContext
7  import de.th_koeln.nt.fowler.fowler.State
8  import de.th_koeln.nt.fowler.fowler.Statemachine
9
10 class FowlerGenerator extends AbstractGenerator {
11
12    override void doGenerate(Resource resource, IFileSystemAccess2 fsa,
    ↪  IGeneratorContext context) {
13       fsa.generateFile("fsm.dot", graphviz(resource))
14    }
15
16    def String graphviz(Resource resource) {
17       '''
18       digraph finite_state_machine {
19           size="8,5"
20
21
22
23           node [shape = record, style = rounded];
24           «FOR state :
    ↪  resource.allContents.toIterable.filter(typeof(State))»
25           «state.name»[label="{«state.name»|«FOR action :
    ↪  state.actions»«action.code» «ENDFOR»}"];
26           «ENDFOR»
27
28           «FOR state :
    ↪  resource.allContents.toIterable.filter(typeof(State))»
29           «FOR transition : state.transitions.filter[
30              !
                 ↪  resource.allContents.toIterable.filter(typeof(Statemachine))
31              .last.resetEvents.contains(it.event)
32           ]»
33           «state.name» -> «transition.state.name» [ label =
    ↪  "«transition.event.name»" ];
34           «ENDFOR»
35           «ENDFOR»
36
37       node [shape = box, style = filled, color=azure2]
38       resetEvents[label="Reset Events:\n«FOR resetEvent :
    ↪  resource.allContents.toIterable
39           .filter(typeof(Statemachine))
40           .last.resetEvents»\n«resetEvent.name»«ENDFOR»"]
41
42       }
43       '''
44    }
45 }
```


Listing 27: Graphviz-Generator, zweite Erweiterung

```
1  /*
2   * generated by Xtext 2.12.0
3   */
4  package de.th_koeln.nt.fowler.tests.ast
5
6  import com.google.inject.Inject
7  import de.th_koeln.nt.fowler.fowler.Statemachine
8  import org.eclipse.xtext.testing.InjectWith
9  import org.eclipse.xtext.testing.XtextRunner
10 import org.eclipse.xtext.testing.util.ParseHelper
11 import org.junit.Assert
12 import org.junit.Test
13 import org.junit.runner.RunWith
14 import org.eclipse.xtext.testing.validation.ValidationTestHelper
15
16 import static
   ↪  de.th_koeln.nt.fowler.tests.ContainsEObject.containsEObject
17 import static org.hamcrest.MatcherAssert.assertThat
18 import static org.hamcrest.CoreMatchers.not
19 import de.th_koeln.nt.fowler.tests.FowlerInjectorProvider
20 import de.th_koeln.nt.fowler.fowler.FowlerFactory
21
22 @RunWith(typeof(XtextRunner))
23 @InjectWith(typeof(FowlerInjectorProvider))
24 class FowlerSimpleParsingTest {
25    @Inject extension ParseHelper<Statemachine>
26    @Inject extension ValidationTestHelper
27
28    @Inject FowlerFactory astFactory
29
30    @Test
31    def void loadModel() {
32       '''
33          events
34             reset R0
35          end
36       '''.parse => [
37          Assert.assertNotNull("Das Result ist nicht null.", it)
38          Assert.assertTrue("Beim Parsen sind Fehler aufgetreten.",
            ↪  it.eResource.errors.isEmpty)
39       ]
40    }
41
42    @Test
43    def void errorsAreFound() {
44       '''
45          wrong text
46       '''.parse => [
47          Assert.assertNotNull("Das Result ist nicht null.", it)
```

```
48          assertThat("Beim Parsen sind keine Fehler aufgetreten.",
            ↪  not(it.eResource.errors.isEmpty))
49       ]
50    }
51
52
53    @Test
54    def void eventsAreCreated() {
55       '''
56          events
57             reset R0
58          end
59       '''.parse => [
60          Assert.assertNotNull("Das Result ist null.", it)
61          assertNoErrors
62          assertThat("Event is not created", it.eAllContents,
            ↪  containsEObject(astFactory.createEvent => [code="R0"
            ↪  name="reset"]))
63       ]
64    }
65
66    @Test def void statesAreCreated() {
67       '''
68          state S0
69          end
70       '''.parse => [
71          Assert.assertNotNull("Das Result ist null.", it)
72          assertNoErrors
73          assertThat("AST is empty.", not(it.eAllContents.empty))
74          assertThat("State is not created", it.eAllContents,
            ↪  containsEObject(astFactory.createState => [name="S0"]))
75       ]
76    }
77
78    @Test def void commandsAreCreated() {
79       '''
80          commands
81             command1 C1
82          end
83       '''.parse => [
84          Assert.assertNotNull("Das Result ist null.", it)
85          assertNoErrors
86          assertThat("AST is empty.", not(it.eAllContents.empty))
87          assertThat("State is not created", it.eAllContents,
            ↪  containsEObject(astFactory.createCommand =>
            ↪  [name="command1" code="C1"]))
88       ]
89    }
90 }
```

Listing 28: FowlerSimpleParsingTest.xtend

```
1  /*
2   * generated by Xtext 2.12.0
3   */
4  package de.th_koeln.nt.fowler.tests.ast
5
6  import com.google.inject.Inject
7  import de.th_koeln.nt.fowler.fowler.FowlerFactory
8  import de.th_koeln.nt.fowler.fowler.Statemachine
9  import de.th_koeln.nt.fowler.tests.FowlerInjectorProvider
10 import org.eclipse.xtext.testing.InjectWith
11 import org.eclipse.xtext.testing.XtextRunner
12 import org.eclipse.xtext.testing.util.ParseHelper
13 import org.eclipse.xtext.testing.validation.ValidationTestHelper
14 import org.junit.Assert
15 import org.junit.Test
16 import org.junit.runner.RunWith
17
18 import static
   ↪  de.th_koeln.nt.fowler.tests.ContainsEObject.containsEObject
19 import static org.hamcrest.CoreMatchers.not
20 import static org.hamcrest.MatcherAssert.assertThat
21
22 @RunWith(typeof(XtextRunner))
23 @InjectWith(typeof(FowlerInjectorProvider))
24 class FowlerComplexParsingTest {
25    @Inject extension ParseHelper<Statemachine>
26    @Inject extension ValidationTestHelper
27
28    @Inject FowlerFactory astFactory
29
30    def String trafficLightControl() {
31       '''
32          events
33             buttonPressed BP
34             timeExpired TE
35             waited W
36          end
37
38          resetEvents
39             timeExpired
40          end
41
42          commands
43             carLightRed CLR
44             carLightYellow CLY
45             carLightGreen CLG
46
47             pedestrianLightRed PLR
48             pedestrianLightGreen PLG
49
50             start4Sec S4S
```

```
51              start20Sec S20S
52          end
53
54          state idle
55              actions {carLightGreen pedestrianLightRed}
56              buttonPressed => wait1
57          end
58
59          state wait1
60              actions{carLightYellow pedestrianLightRed start4Sec}
61              waited => idle
62          end
63      '''
64  }
65
66  @Test def void commandsAreCreated() {
67      trafficLightControl.parse => [
68          Assert.assertNotNull("Das Result ist nicht null.", it)
69          assertNoErrors
70          assertThat("AST is not empty.", not(it.eAllContents.empty))
71          assertThat("State is created", it.eAllContents,
            ↪  containsEObject(astFactory.createCommand => [
72              name = "carLightRed"
73              code = "CLR"
74          ]))
75          assertThat("State is created", it.eAllContents,
            ↪  containsEObject(astFactory.createCommand => [
76              name = "carLightYellow"
77              code = "CLY"
78          ]))
79          assertThat("State is created", it.eAllContents,
            ↪  containsEObject(astFactory.createCommand => [
80              name = "carLightGreen"
81              code = "CLG"
82          ]))
83          assertThat("State is created", it.eAllContents,
            ↪  containsEObject(astFactory.createCommand => [
84              name = "pedestrianLightRed"
85              code = "PLR"
86          ]))
87          assertThat("State is created", it.eAllContents,
            ↪  containsEObject(astFactory.createCommand => [
88              name = "pedestrianLightGreen"
89              code = "PLG"
90          ]))
91          assertThat("State is created", it.eAllContents,
            ↪  containsEObject(astFactory.createCommand => [
92              name = "start4Sec"
93              code = "S4S"
94          ]))
```

```
95            assertThat("State is created", it.eAllContents,
              ↪  containsEObject(astFactory.createCommand => [
96               name = "start20Sec"
97               code = "S20S"
98            ]))
99         ]
100     }
101
102     @Test def void eventsAreCreated() {
103        trafficLightControl.parse => [
104           Assert.assertNotNull("Das Result ist nicht null.", it)
105           assertNoErrors
106           assertThat("AST is not empty.", not(it.eAllContents.empty))
107           assertThat("Event is created", it.eAllContents,
              ↪  containsEObject(astFactory.createEvent => [
108              name = "buttonPressed"
109              code = "BP"
110           ]))
111           assertThat("Event is created", it.eAllContents,
              ↪  containsEObject(astFactory.createEvent => [
112              name = "timeExpired"
113              code = "TE"
114           ]))
115           assertThat("Event is created", it.eAllContents,
              ↪  containsEObject(astFactory.createEvent => [
116              name = "waited"
117              code = "W"
118           ]))
119        ]
120     }
121
122     @Test def void resetEventsAreCreated() {
123        trafficLightControl.parse => [
124           Assert.assertNotNull("Das Result ist nicht null.", it)
125           assertNoErrors
126           assertThat("AST is not empty.", not(it.eAllContents.empty))
127           assertThat("ResetEvent is created", it.resetEvents.iterator,
              ↪  containsEObject(astFactory.createEvent => [
128              name = "timeExpired"
129              code = "TE"
130           ]))
131        ]
132     }
133  }
```

Listing 29: FowlerComplexParsingTest.xtend

Erklärung über die selbständige Abfassung der Arbeit

Ich erkläre an Eides statt, dass ich die vorgelegte Abschlussarbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

(Ort, Datum, Unterschrift)

Errata

Diese Seite war nicht Teil der am 15.10.2018 eingereichten Masterarbeit. In den vergangenen Monaten konnten allerdings die beiden in Abschnitt 6.1 beschriebenen Herausforderungen geklärt werden.

Der Einsatz von *AssertJ* hat sich als nicht zielführend herausgestellt. Vor allem lies sich das Framework nicht so einfach für die Ecore-Klassen erweitern wie Hamcrest.

Andererseits konnte eine Lösung gefunden werden, wie das sehr unsaubere lokale Einbinden der Hamcrest-Library umgangen werden kann. Dazu muss in der Eclipse Target Definition das Paket *org.hamcrest.library* aus dem passenden Orbit-Repository eingebunden werden (siehe Listing 30). Auf diese Weise wird es sowohl vom Eclipse-Testrunner als auch von Maven gefunden, welches im Xtext-Kontext das Tycho Surefire Plugin und damit indirekt Eclipse zum Ausführen der Tests nutzt.

Nachvollziehen lassen sich die nötigen Änderungen im Github-Projekt unter *https://github.com/martinschulze/failing-maven-build-example* im Commit *281ebaa* vom 08.01.2020. Der Commit ist leider etwas unübersichtlich geraten, da gleichzeitig die Umstellung auf eine aktuellere Eclipse-Version (2019.3) sowie Junit 5 vollzogen wurde, um die Probleme mit Junit 4 und Hamcrest zu umgehen.

```
3  <target name="de.th_koeln.nt.fowler.target" sequenceNumber="1">
4  <locations>
...
38 <location includeAllPlatforms="false" includeConfigurePhase="false"
   ↪ includeMode="planner" includeSource="true"
   ↪ type="InstallableUnit">
39 <unit id="org.hamcrest.library" version="0.0.0"/>
40 <repository location="http://download.eclipse.org/tools/orbit/
       downloads/drops/R20190226160451/repository"/>
41 </location>
42
43 </locations>
44 </target>
```

Listing 30: Hamcrest-Library in der Eclipse Target Definition